Saimaan ammattikorkeakoulu
Tekniikka Lappeenranta
Tietotekniikka
Tietojärjestelmien kehitys

Petri Miikki

Trade Blotter -ohjelma

Opinnäytetyö 2014

Tiivistelmä

Petri Miikki
Trade Blotter -ohjelma, 62 sivua
Saimaan ammattikorkeakoulu
Tekniikka Lappeenranta
Tietotekniikan koulutusohjelma
Tietojärjestelmien kehitys
Opinnäytetyö 2014
Ohjaajat: lehtori Martti Ylä-Jussila, Saimaan ammattikorkeakoulu,
ohjelmistonkehitysjohtaja Sampo Lankinen, FinFX Trading Oy

Tämän opinnäytetyön tarkoituksena oli suunnitella, toteuttaa ja testata ohjelma, joka seuraa FinFX Trading Oy:n asiakkaiden tekemiä kauppoja reaaliaikaisesti. Ohjelman tarkoituksena on helpottaa FinFX Trading Oy:n jokapäiväistä liiketoimintaa ja pienentää mahdollisia taloudellisia riskejä, jotka johtuisivat heidän asiakkaidensa toimista.

Ohjelmasta ei ollut tarkkaa vaatimusmäärittelyä, vaan sen tekemiseen käytettiin ketterää kehitysmenetelmää, jotta ohjelmasta saatiin mahdollisimman nopeasti asiakkaalle testattavia versioita. Ohjelman toimintalogiikan toteuttamiseen käytettiin C#-ohjelmointikieltä sekä Model-View-ViewModel-arkkitehtuuria, käyttöliittymien luomiseen käytettiin Windows Presentation Foundation-komponenttikirjastoa ja tietokantayhteyksiin Entity Framework-teknologiaa.

Opinnäytetyön tuloksena asiakkaalle luovutettiin luotettavasti sekä vaatimusten mukaisesti toimiva ohjelma. Ohjelman ansiosta asiakkaalla on mahdollista välittömästi saada tietoa asiakkaiden tekemistä kaupoista.

Avainsanat: C#, olio-ohjelmointi, ohjelmointi, WPF, XAML, Entity Framework

Abstract

Petri Miikki
Trade Blotter - program, 62 Pages
Saimaa University of Applied Sciences
Technology Lappeenranta
Information Technology Degree Programme
Development of Information Systems
Bachelor´s Thesis 2014
Instructors: lecturer Martti Ylä-Jussila, Saimaa University of Applied Sciences
software development manager Sampo Lankinen FinFX Trading Oy

The purpose of this thesis was to design, implement and test a program that will monitor trades made by customers of FinFX Trading Oy in real time. The purpose of the program is to make FinFX Trading Oy's daily business easier and also reduce possible financial risks caused by their customers' trades.

The program did not have requirement specification. It was developed using agile software development, so the client of the thesis would get faster new versions of the program to test. The business logic of the program was made in object-oriented programming language C# with Model-View-ViewModel architecture, graphical user interfaces were made using Windows Presentation Foundation and database connections were made using Entity Framework.

As the result of this thesis, the customer got a reliably working program that fulfills all the requirements given. Because of the program, the customer can immediately know trades that their customers have made without delays.

Keywords: C#, object-oriented programming, programming, WPF, XAML, Entity Framework

Sisältö

Termit ja käsitteet

.NET Framework	.NET Framework on Microsoftin kehittämä ohjelmisto-komponenttikirjasto.
Apache	Apache Software Foundation on voittoa tavoittelematon yritys, jolla on kehityksessä yli 140 vapaan lähdekoodin projektia.
API	Application Programming Interface eli ohjelmointirajapinta.
C	C on yleiskäyttöinen ohjelmointikieli, jonka kehitti Dennis Ritchie 1970-luvun alussa.
C#	C# (luetaan c sharp) on Microsoftin kehittämä olio-ohjelmointikieli.
C++	C++ on C-ohjelmointikielen pohjalta luotu ohjelmointikieli, johon lisätty muun muassa olio-ohjelmoinnin mahdollistavat toiminnot.
CLR	Common Language Runtime on MSIL-kielen ajoympäristö, joka suorittaa koodia ja tarjoaa palveluita, jotka tekevät kehitysprosessista helpomman.
DMA	Direct Market Access tarjoaa kauppiaille suoran yhteyden oikeisiin markkinapaikkoihin.
ECN	Electronic Communication Network kuvaa sähköiskäfoorumia tai tietoverkkoa, joka helpottaa finanssituotteiden kauppaa perinteisen osakkeidenvaihdon ulkopuolella.
Entity FrameWork	Entity Framework on ORM-kehys.
Fontti	Fontti eli kirjasintyyppi on painetun tai sähköisesti tuotetun tekstin ulkonäköstandardi.
HTML	Hyper Text Markup Language on merkintäkieli, joita voidaan näyttää Internet-sivuina.
Interface	Rajapinta, joka jakaa rajan kahden eri komponentin kanssa.
Java	Java on korkeantason olio-ohjelmointikieli, jonka kehitti Sun Microsystems.
LINQ	Language-Integrated Query on ohjelmointikieleen upotettu kyselykieli, joka laajentaa C#- ja Visual Basic-ohjelmointikielien kyselymahdollisuuksia.

Log4net	Log4net on apuväline, joka auttaa ohjelmoijaa kirjoittamaan ohjelman tilatietoja useisiin eri ulostulokohteisiin.
Metadata	Metadata on tietoa, joka kuvailee muuta tietoa.
Metodi	Metodi on jäsenaliohjelma, joka käsittelee luokan tietoa.
Olio	Olio-ohjelmoinnissa olio on luokan ilmentymä.
Parametri	Parametrilla välitetään tietoa metodille.
Prosessi	Tietokonealalla prosessi tarkoittaa tietokoneohjelman ilmentymää, jota suoritetaan.
Renderöinti	Renderöinti on prosessi, jossa luodaan kuva mallista.
Silverlight	Silverlight on Microsoftin kehittämä teknologia, joka on kehitetty luomaan Internet- ja mobiilisovellusten käyttöliittymiä.
SmallTalk	SmallTalk on 1970-luvulla julkaistu olio-ohjelmointikieli.
Säie	Säikeellä prosessi voidaan jakaa useampaan samanaikaisesti suoritettavaan tehtävään.
UML	Unified Modeling Language on yleishyödyllinen mallinnuskieli ohjelmistokehityksen alalla.
UNIX	UNIX on käyttöjärjestelmä.
XAML	Extensible Application Markup Language on XML-kielen murre, jota käytetään WPF ja Silverlight teknologioissa käyttöliittymien luomiseen.
XML	Extensible Markup Language on suunniteltu olemaan itseään kuvaava merkintäkieli, joka on ihmisluettavassa muodossa.
Visual Basic	Visual Basic on Microsoftin kehittämä olio-ohjelmointikieli.
WPF	Windows Presentation Foundation on Microsoftin kehittämä teknologia työpöytäsovellusten käyttöliittymien luomista varten.

1 Johdanto

Opinnäytetyön tavoitteena on suunnitella, toteuttaa ja testata ohjelma, joka pystyy reaaliaikaisesti seuraamaan ja esittämään FinFX Trading Oy:n asiakkaiden tekemiä valuuttakauppoja. Ennen opinnäytetyötä FinFX Trading Oy:llä ei ollut tehokasta ja nopeaa asiakkaiden kauppojenseurausjärjestelmää, jonka vuoksi he eivät pystyneet nopeasti reagoimaan asiakkaan kauppoihin. Opinnäytetyön tarkoituksena on yksinkertaistaa ja helpottaa tätä tehtävää.

Ohjelman arkkitehtuurin vaatimuksena on, että ohjelmaan voi tulevaisuudessa helposti lisätä uusia ominaisuuksia ilman, että ohjelman arkkitehtuuria joutuisi muokkaamaan. Ohjelma toteutetaan täysin täysin Microsoftin kehittämillä teknologioilla, koska FinFX Trading Oy:n muu ohjelmistonkehitys tehdään kyseisillä teknologioilla. Toteutusmuodoksi on valittu työpöytäsovellus, johon käyttöliittymä toteutetaan Windows Presentation Foundation-kirjaston komponenteilla.

FinFX Trading Oy

FinFX Trading Oy on Imatralla sijaitseva ECN/DMA valuuttameklariyritys, joka on erikoistunut online-valuutta- ja raaka-ainekaupan välittämiseen. Yritys on kasvanut nopeasti perustamisvuodesta 2010 lähtien, jolloin yritys työllisti vain muutaman henkilön, kun vuonna 2014 FinFX Trading Oy:n palveluksessa on jo noin 30 työntekijää. Yrityksen asiakaskunta koostuu tuhansista yksityis- ja yritysasiakkaista yli sadasta eri maasta. (FinFX Trading Oy.)

2 Ketterä ohjelmistokehitys

Ketterät ohjelmistokehitystavat ovat vastareaktio perinteisille raskaille suunnitelmaohjatuille kehitysprosesseille. Perinteisissä kehitysprosesseissa työ alkaa täydellisen vaatimusmäärittelyn dokumentoinnista, jota seuraa arkkitehtuuri- ja korkeantasonsuunnittelu, kehitys ja tarkastaminen. Teollisuuden ja teknologian nopean kehityksen takia asiakkaiden kyky määritellä etukäteen täydellisesti haluamaansa tuotteet on vaikeutunut. (DACS, Agile Software Development, s. 2.)

Tämän seurauksena useat konsultit ovat itsenäisesti kehittäneet menetelmiä ja käytäntöjä vastaamaan väistämätöntä muutosta, joka oli tapahtumassa. Nämä ketterät menetelmät eivät itse asiassa esittäneet uusia menetelmiä, vaan olivat joukko eri käytäntöjä, jotka jakoivat samat arvot ja käytännöt. Monet näistä käytännöistä esimerkiksi perustuvat iteraatioihin, jotka esiteltiin jo vuonna 1975. (DACS, Agile Software Development, s. 2 -3.)

Ennen vuotta 2000 oli ehditty jo julkaisemaan lukuisia kevyisiin iteraatioihin perustuvia ohjelmistonkehitysmenetelmiä. Näistä eniten huomiota vuosien saatossa on kerännyt Kent Beckin kehittämä XP-kehitysmenetelmä (eXtreme Programming). Ketteriä kehitysmenetelmiä edistämään perustettiin vuonna 2001 Agile Alliance -järjestö, joka julkisti oman manifestinsa, Agile Manifesto, ohjaamaan ketteriä menetelmiä (Haikala & Mikkonen 2011, s. 43 – 44):

We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

- Individuals and interactions over processes and tools
- Working software over comprehensive documentation
- Customer collaboration over contract negotiation
- Responding to change over following a plan

That is, while there is value in the items on the right, we value the items on the left more. (Agile Manifesto, Manifesto for Agile Software Development.)

Manifestin pääsanomana on, että asiakkaan tyytyväisyys ja ohjelman toimivuus on tärkeämpää kuin laaja dokumentaatio ja etukäteissuunnittelu. Manifesti korostaa yhteydenpidon merkitystä asiakkaan kanssa: on pystyttävä nopeasti reagoimaan, jos muutostarpeita ilmentyy kehitystyön aikana.

Agile Manifesto määrittelee kaksitoista periaatetta, joita ketterissä kehitysmenetelmissä tulisi noudattaa. Yksi tärkeimmistä periaatteista on asiakastyytyväisyys ja jatkuva uusien ohjelmistoversioiden toimittaminen asiakkaalle koko projektin ajan. (Agile Manifesto, Principles.)

3 Arkkitehtuuri

Ohjelmistoarkkitehtuuri tarkoittaa yksinkertaisimmillaan ohjelman komponenttien sekä niiden välisiä suhteita. Melkein aina arkkitehtuuriin ajatellaan myös

kuuluvan suuntaviivat jatkokehitykselle ja vähintään pieni kuvaus käytettävistä suunnitteluperiaatteista. (Haikala & Mikkonen 2011, s. 178.)

Toinen määritelmä ohjelmistoarkkitehtuurista on, että se kuuluu siihen osaan ohjelmistoa, joka ei tule muuttumaan ohjelman ylläpidon aikana. Näin arkkitehtuuri tarjoaa mahdollisuudet johdonmukaiselle ohjelmistonkehitykselle, testaukselle sekä ylläpidolle, koska siihen niiden aikana yleensä kohdistu suuria muutoksia. (Haikala & Mikkonen 2011, s. 178.)

Arkkitehtuuri korostuu ohjelmiston tärkeänä osa-alueena, koska siinä määritelty rakenne tulee toimimaan koko ohjelmiston pohjana. Arkkitehtuuriset ongelmat ohjelmassa ovat yleensä aina vakavia, koska ne tulevat vaikuttamaan ohjelmankehityksen suunnitteluvaiheen jälkeisiin vaiheisiin. Huono arkkitehtuuri voi aiheuttaa ohjelman kehityksessä suuriakin lisäkustannuksia, jotka voivat johtua esimerkiksi suorituskykyongelmista tai ohjelma ei ole hyvin laajennettavissa, joka vaatii paljon turhaa lisätyötä. (Haikala & Mikkonen 2011, s. 178 – 179.)

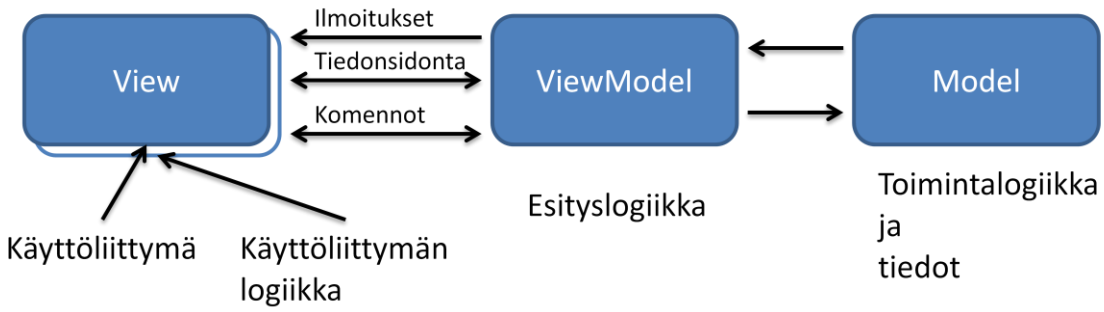
3.1 Model-View-ViewModel

Siitä saakka, kun ihmiset ovat luoneet ohjelman käyttöliittymiä, niin niiden luomisessa on hyödynnetty erilaisia suunnittelumalleja helpottamaan työtä. Vuonna 2005 Microsoftin Windows Presentation Foundationin ja Silverlightin arkkitehti, John Gossman, julkaisi Model-View-ViewModel-mallin (MVVM). (MSDN Magazine, WPF MVVM.)

MVVM-malli auttaa erottamaan selkeästi ohjelman toimintalogiikan sen käyttöliittymästä. Näiden kahden asian erottaminen selkeästi toisistaan auttaa monissa kehityksen ja suunnittelun vaiheissa sekä tekee ohjelman testauksesta, ylläpidosta ja jatkokehityksestä huomattavasti helpompaa. Mallin avulla pystyy myös helpommin uudelleen käyttämään koodia ja se mahdollistaa ohjelmistokehittäjien ja käyttöliittymäsuunnittelijoiden sujuvan yhteistyön ohjelmistokehityksen aikana. (Microsoft, Implementing the MVVM Pattern.)

Käyttämällä MVVM-mallia ohjelman käyttöliittymä, käyttöliittymän tiedot ja toimintalogiikka jaetaan kolmeen eri luokkaan: View, Model ja ViewModel. View-luokat kapseloivat ohjelman käyttöliittymän ja sen logiikan, Model-luokat kapse-

loivat ohjelman toimintalogiikan sekä ohjelmassa käytettävän tiedot ja ViewModel-luokat kapseloivat esityslogiikan ja –tilan (Microsoft, Implementing the MVVM Pattern.). Kuva 1 näyttää näiden kolmen MVVM-luokan vuorovaikutuksen niiden kesken.



Kuva 1. MVVM-mallin luokkien vuorovaikutus.

Model-luokat ovat vuorovaikutuksessa ViewModel-luokkien kanssa, joista tieto sidotaan edelleen käyttöliittymän View-luokkiin. Myös ViewModel-luokissa olevat komennot voidaan sitoa käyttöliittymän elementteihin, jolloin käyttöliittymän elementtejä voi helposti vaihtaa uusiin ja sitoa sama toiminnallissuun uuteen elementtiin.

4 Olio-ohjelmointi

Oliosuuntautunut ohjelmointi (Object-oriented programming OOP) on lähestymistapa ohjelmistokehitykseen, jossa ohjelman rakenne perustuu olioiden keskinäiseen vuorovaikutukseen saavuttaakseen vaadittavan tehtävän. Vuorovaikutuksen muoto on viestien lähettäminen olioiden välillä. Vastauksena johonkin viestiin olio voi suorittaa siltä vaaditun toiminnon. (Clark 2010, s. 1.)

4.1 Historia

Olio-ohjelmoinnin konsepti alkoi kehittyä 1960-luvun puolivälissä Simula-ohjelmointikielellä (Clark 2010, s. 1). Simula-ohjelmointikieltä pidetään ensimmäisenä olio-ohjelmointikielenä, joka esitteli ohjelmoijille muun muassa seuraavat käsitteet: oliot, luokat ja perinnän. Muun muassa suosittu C++-ohjelmointikieli on syntynyt Simula-kielen mekanismeista, jotka inspiroivat tans-

kalaista Bjarne Stroustrupia laajentamaan UNIX-pohjaista C-ohjelmointikieltä. (Birth of OO, The Simula Languages)

Olio-ohjelmoinnin konseptit kehittyivät edelleen 1970-luvulla, kun uusi ohjelmointikieli SmallTalk teki tuloaan. Vaikka ohjelmistokehittäjät eivät alun perin olleet erityisen innostuneista uusista mahdollisuuksista, jotka olio-ohjelmointi toi tullessaan, niin silti olio-ohjelmoinnin konseptit jatkoivat kehittymistään. 1980-luvun puolivälissä kiinnostus olio-ohjelmoinnin konsepteihin alkoi elpyä, kun esimerkiksi C++-olio-ohjelmointikieli tuli suosituksi ohjelmoijien enemmistön keskuudessa. (Clark 2010, s. 2.)

1990-luvulla olio-ohjelmoinnin suosio jatkoi kasvuaan, kun uusi ohjelmointikieli, Java, ilmestyi. Vuonna 2002 Microsoft julkaisi myös uuden olio-ohjelmointikielen, C#:n ja uudisti Visual Basic-ohjelmointikielen tukemaan olio-ohjelmointia tehden siitä myös oikean olio-ohjelmointikielen (Clark 2010, s. 2). Kolme suosituinta olio-ohjelmointikieltä ovat huhtikuussa 2014 järjestyksessään: Java, Objective-C ja C++ (TIOBE, Index for April 2014.).

4.2 Oliot

Olio-ohjelmoinnin käsitteissä olio on rakenne, joka sisältää tietoa ja toimintoja tietojen käyttämiseen (Clark 2010, s. 3), eli olio on luokan ilmentymä. Jokainen luokasta luotu olio tulee kehittymään sen oman tietojen käyttämisen mukaan, mutta se jakaa aina saman toiminnallisuuden muiden samasta luokasta luotujen olioiden kanssa (Champlain & Patrick 2005, s. 10).

Voidaan katsoa, että maailma koostuu olioista. Esimerkiksi kaikilla ihmisillä on yhteisiä tietoja, kuten esimerkiksi pituus, paino ja silmien väri. Ihmiset ovat myös tekemisessä toistensa kanssa, jolloin tapahtuu olioiden välinen vuorovaikutus. Kaikilla ihmisillä on myös yhteisiä toimintoja, esimerkiksi syöminen ja hengittäminen. Näiden toimintojen suorittaminen voi riippua ihmisen henkilökohtaisista tiedoista, mutta toiminnallisuus pysyy kuitenkin samana muiden ihmisen kanssa. (Clark 2010, s. 3.)

4.3 Abstraktio

Elämä on täynnä monimutkaisia asioita, joista jokaisen käsitteleminen yksikkönä on useasti liian vaikeaa. Esimerkiksi ihminen koostuu useammasta kuin oktiljoonasta (10 potenssiin 48) atomista. Ihminen mielletään kuitenkin yhdeksi kokonaisuudeksi, eikä käsitellä joukkona eri ominaisuuksia, koska se on huomattavasti helpompi ymmärtää. (Prata 2012, s. 507.)

Ohjelmoinnissa abstraktio on erittäin tärkeä askel olion sisäisen tiedon esittämiseen rajapinnan (interface) ehdoilla (Prata 2012, s. 507). Julkinen rajapinta suunnittelukomponenttien abstraktiota (Prata 2012, s. 512). Luokat vähentävät monimutkaisuutta seuraavilla abstraktion tavoille (Champlain & Patrick 2005, s. 10):

- piilottamalla toteutuksen tiedot
- korostamalla tärkeää toimintaa luokan käyttäjän osalta (käyttöliittymä)
- erottamalla käyttöliittymän ja toteutuksen toisistaan

4.4 Kapselointi

Luokan sisäisen tiedon ja toiminnallisuuden piilottaminen on kapseloinnin ilmentymä. Toisin sanoen kapselointi tarkoittaa, että luokan täytäntöönpano tiedot ovat piilotettu, kuten tiedon esittäminen ja toiminnallisuuden koodi (Prata 2012, s. 512, 1348).

Jos luokan käyttäjä haluaa pääsyn luokan tietoihin, niin hänen tulee tehdä se luokan julkisen rajapinnan kautta. Tiedon kapselointi tekee luokan käytön turvallisemmaksi ja luotettavammaksi, koska luokan suunnittelija tietää, millä tavalla tietoa voi käyttää ja mitä toimintoja voi suorittaa luokan tietoihin. Kapselointi helpottaa ohjelman ylläpitoa huomattavasti, koska se mahdollistaa luokan toiminnallisuuden muokkaamisen ja parantamisen ilman, että se vaikuttaa luokan käyttöön. (Clark 2010, s. 4.)

Luokan ylläpitäjä voi keksiä uuden tehokkaamman ja nopeamman tavan toiminnon toteuttamiseksi, jolloin hän voi muokata kyseistä toimintoa. Koska käyttöliit-

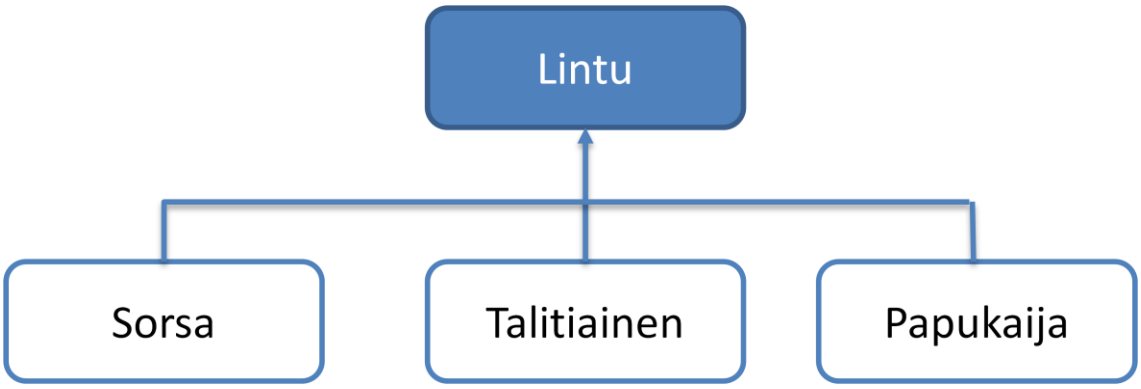
tymä tulee pysymään luokalle samana, niin luokan käyttäjän ei tarvitse muokata oman ohjelmansa koodia, vaikka luokkaa on muokattu.

4.5 Periytyminen

Yksi olio-ohjelmoinnin päämääristä on mahdollistaa koodin uudelleen käyttäminen. Vanhan koodin uudelleen käyttäminen säästää aikaa ja auttaa vähentämään uusien virheiden ilmentymästä ohjelmaan, koska koodi on aikaisemmin testattu ja käytetty. (Prata 2012, s. 707.)

Periytyminen sallii vanhojen luokkien uusiokäytön, kun niistä voi periyttää uusia luokkia. Periytetty luokka perii kaikki ominaisuudet, esimerkiksi luokan toiminnallisuuden vanhasta luokasta, jota kutsutaan kantaluokaksi. Luokkien periyttäminen jo olemassa olevista luokista on yleensä helpompaa kuin suunnitella kokonaan uusi luokka. (Prata 2012, s. 708.)

Periytymistä käytetään määrittelemään kantaluokkia, joihin kuuluu yhteisiä tietoja ja toimintoja (Clark 2010, s. 5). Näin useampaan luokkaan kuuluvat tiedot voidaan kerätä yhteen luokkaan, jota sitten käytetään periyttämisessä kantaluokkana. Tämä mahdollistaa, että samaa koodia ei tarvitse kirjoittaa useampaan kertaan ja kantaluokan muutokset vaikuttavat suoraan kaikkiin siitä periytyviin luokkiin. Kuva 2 esittää oliokaaviota, jossa on kantaluokka Lintu.



Kuva 2. Luokkakaavio periytymisestä olio-ohjelmoinnissa.

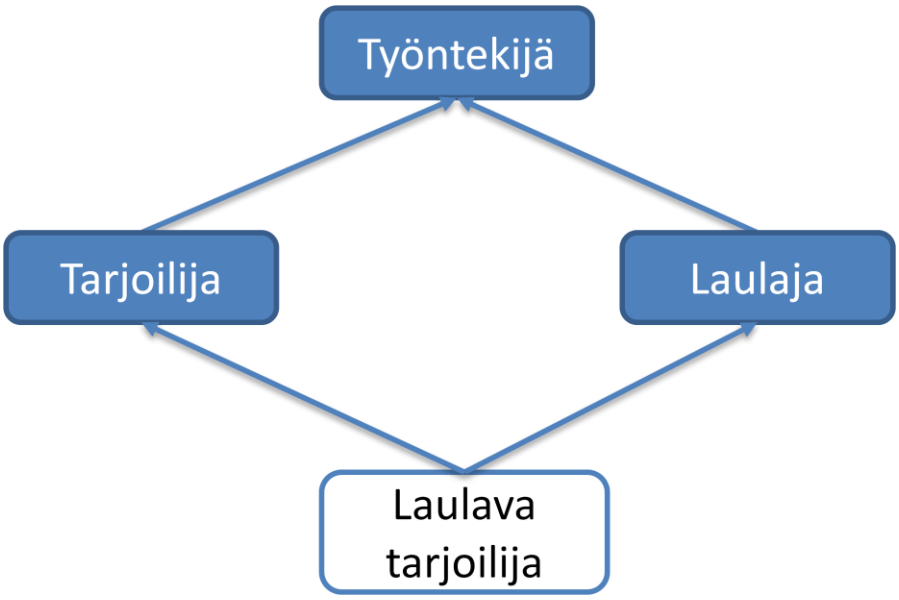
Lintu-luokasta periytyvät muut luotavat linnut, jolloin kaikki lintujen yhteiset ominaisuudet, jotka ovat jo määritelty kantaluokassa tulevat myös käytettäviksi siitä periytyneille luokille. Tämä mahdollistaa myös lintujen erilaisuuden, koska pe-

riytetyissä luokissa voidaan määritellä vielä uusia ominaisuuksia linnuille tai muuttaa toiminnallisuutta enemmän uudelle linnulle sopivammaksi.

Yksi periytyminen lisäksi osa ohjelmointikielistä, kuten C++, mahdollistaa myös moniperiytymisen. Moniperiytyminen aiheuttaa ohjelmoijille uusia ongelmia, joista kaksi keskeisintä ongelmaa ovat kahden samannimisen toiminnallisuuden periytyminen kahdesta eri kantaluokasta ja saman luokan instanssin usean periytyminen monesta luokasta. Näiden ongelmien ratkominen vaatii uusien sääntöjen ja syntaksien käyttämistä, jonka takia moniperiytyminen enemmän virhealtista kuin yksi periytymisen käyttäminen. (Prata 2012, s. 809.)

Moniperiytyminen on aiheuttanut paljon keskustelua vastaan ja puolesta. Osan mielestä kyseinen ominaisuus tulisi jopa poistaa kokonaan, kun osan mielestä moniperiytyminen on erittäin hyödyllistä ja osissa projekteista jopa pakollista. Tämän lisäksi on vielä näkemys, että moniperiytymistä tulisi käyttää vain varovaisesti ja maltillisesti (Prata 2012, s. 809.). Näiden ongelmien takia useat ohjelmointikielet eivät ollenkaan salli moniperiytymisen käyttöä. Esimerkiksi C# ja Java eivät tarjoa mahdollisuutta moniperiytymiseen, vaan tarjoavat vastaavan toiminnan rajapintojen kautta.

Kuva 3 esittää moniperiytymiseen liittyvän ongelman, joka ratkaistavissa C++-ohjelmointikielessä käyttämällä siihen tarkoitettua syntaksia. Kuvan mukaisessa moniperiytymisessä Laulava tarjoilija -luokka tulee perimään kaksi kertaa Työntekijä-luokan, jolloin se tulee kahteen kertaan sisältämään Työntekijä-luokan tiedot.



Kuva 3. Moniperiytymisen diamond of death -ongelma.

4.6 Polymorfismi

Polymorfismi eli monimuotoisuus on kahden eri olion kyky vastata samaan viestiin niiden omalla toiminnallisuudella (Clark 2010, s. 4). Olio-ohjelmoinnissa, kun periytetään kantaluokasta aliluokkia, niin aliluokissa voidaan uudelleen määritellä kantaluokan toiminallisuutta. Tämä mahdollistaa monimuotoisuuden toteutumisen ja toiminnallisuuden ajonaikaisen valinnan eli metodien dynaamisen sidonnan. (Prata 2012, s. 722 – 726.)

5 Tekniikat

Asiakkaan esittämien yhteensopivuus- ja ylläpidettävyysvaatimusten takia opinnäytetyö tehtiin täysin Microsoftin kehittämillä teknologioilla, jotka mahdollistivat nopean ja tehokkaan kehittämisaikataulun. Tässä luvussa käydään läpi opinnäytetyössä käytetyt tekniikat.

5.1 Windows Presentation Foundation

Windows Presentation Foundation (WPF) on Microsoftin tärkein teknologia graafisten käyttöliittymien luontiin. WPF:llä pystyy luomaan yksinkertaisia lomakkeita, dokumenttikeskeisiä ikkunoita, videoita, 2D-grafiikkaa sekä myös monimutkaisia 3D-grafiikkaympäristöjä. Windows Presentation Foundation tar-

joaa työkalut, joilla helppo luoda laaja-alaisesti runsasominaisuuksisia käyttöliittymiä. WPF toimii myös perustana Microsoftin luomalle Silverligth-tekniikalle, joka laajentaa WPF:n teknologiaa Internetin puolelle ja erilaisille laitteille kuten Windows Phonelle. (Nathan 2010.)

WPF:n ytimenä on näytön resoluutiosta riippumaton ja vektoripohjainen mallinnusmoottori, joka on rakennettu hyödyntämään nykyaikaista graafista laitteistoa. WPF laajentaa ydintä monipuolisella joukolla ohjelmistonkehitysominaisuuksia sisältäen muun muassa tyylit, mallit, kontrollit ja Extensible Application Markup Language -kielen (XAML), jota käytetään käyttöliittymien muotoiluun sekä erityisesti käyttöliittymän tiedonsidontaan. WPF sisältyy Microsoftin .NET Frameworkiin, joten ohjelmat pystyvät vaivattomasti käyttämään .NET:n laajaa valmista luokkakirjastoa. (Microsoft, Windows Presentation Foundation.)

Käyttöliittymien luontialustana WPF eroaa edellisistä teknologioista suuresti muun muassa ohjelmointimallissa, taustalla olevissa käsitteissä ja yleisessä terminologiassa (Nathan 2010). WPF tarjoaa lukuisia ohjelmointiparannuksia Windowsin työpöytäsovelluksien kehittämiseen. Yksi oleellinen parannus on mahdollisuus kehittää ohjelman käyttöliittymää XAML-merkkikieltä käyttäen ja käyttää erillistä ohjelmointikieltä ohjelman toimintojen toteuttamiseen, jolloin näkymän ja toiminnan erottaminen toisistaan tarjoavat muun muassa seuraavat hyödyt:

- Kehitys ja ylläpito kulut vähentyvät, koska näkymä ei ole tiukasti kytketty ohjelman toimintaan. Tämä mahdollistaa käyttöliittymän muokkaamisen jälkeen erittäin helposti.
- Antaa suunnittelijoille mahdollisuuden toteuttaa ohjelman käyttöliittymän samanaikaisesti kehittäjien kanssa, jotka toteuttavat ohjelman toiminnallisuutta.

5.1.1 XAML

XAML on deklaratiivinen merkintäkieli, joka on XML-merkintäkielen murre, jota käytetään yleisesti .NET-teknologioissa luomaan toiminnallisuutta deklaratiivisella tavalla (Nathan 2010, s. 21). XAML yksinkertaistaa käyttöliittymien luonnin .NET Framework-sovelluksiin luoden käyttöliittymäelementtejä selittävällä

XAML-merkinnällä. Tämän ansiosta käyttöliittymän määritelmä ja ohjelman ajoaikainen logiikka pystytään erottamaan toisistaan. (Microsoft, XAML.).

XAML:n tarkoitus on helpottaa ohjelmoijien muiden alojen ammattilaisten yhteistyötä, jolloin XAML:sta tulee työnteon yhteinen kieli. Ohjelmistokehitys tapahtuu pääasiassa kehitystyökalujen avulla, mutta XAML on myös yleisesti ihmisten luettavissa olevassa muodossa, jolloin työkaluksi riittää pelkästään tekstieditori. (Nathan 2010, s. 21.)

Olioelementtisyntaksi (object element syntax) on XAML-kielen syntaksi, jolla alustetaan CLR-luokkia tai rakenteita määrittelemällä XML-elementin. Syntaksi muistuttaa hyvin paljon muiden merkintäkielien elementtisyntaksia, kuten esimerkiksi HTML-kielen. Olioelementtisyntaksi alkaa vasemmalla kulmasulkeella (<), jota välittömästi seuraa alustettavan luokan tai rakenteen tyyppinimi. Syntaksin sulkee vinoviiva (/), jota seuraa oikea kulmasulku (>). Vaihtoehtoisesti syntaksin voi sulkea myös *</luokannimi>* -merkintä. Syntaksin alkamisen jälkeen sitä ei ole pakko sulkea heti, vaan se voidaan täydentää oikealla kulmasulkeella, jonka jälkeen siinä voi määritellä esimerkiksi luokan ominaisuuksia ennen syntaksin sulkemista. (Microsoft, Object Element Syntax.)

WPF:ssä kaikki käyttöliittymän elementit, esimerkiksi Button-luokka, on pakko sijoittaa paneelien sisään, joita on olemassa useita erilaisia, joista tärkein on Grid. Grid on WPF:n valmiista paneeleista parhain vaihtoehto monimutkaisiin käyttöliittymien määrittelyyn, koska se pystyy tekemään saman ja enemmän kuin muut paneelit, lukuun ottamatta WrapPanelin käärintä-ominaisuutta (Nathan 2012, s. 136). Seuraavassa esimerkissä luodaan yksinkertainen käyttöliittymä, joka sisältää vain yhden käyttöliittymäelementin, Napin (Button). Kuva 4 esittää alla olevan XAML-merkinnän käyttöliittymän.

```
<Window x:Class="WPF_esimerkki.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        SizeToContent="WidthAndHeight">
    <Grid>
        <Button Content="Klikkaa" />
    </Grid>
</Window>
```

Kuva 4. Yksinkertainen esimerkki XAML:llä luodusta ikkunasta.

Yllä olevan XAML-esimerkin mukaisesti WPF-ikkunan määrittely alkaa aina Window-luokan määrittelyllä, joka luo, säätää, näyttää sekä hallitsee ikkunaa koko sen eliniän. Kuten esimerkistä huomataan, näiden elementtien sisällä voidaan määritellä, niille myös ominaisuuksia, jotka tekevät XAML-kielestä erittäin tehokkaan käyttää.

x:Class-ominaisuus määrittelee luokan, jota *Window*-elementti tulee käyttämään. Ensin määritellään nimiavaruus, johon luokka kuuluu. Nimiavaruuden määritteleminen ei kuitenkaan ole pakollista. Esimerkissä nimiavaruus on *WPF_esimerkki*, jossa *MainWindow*-luokka sijaitsee. Tämän merkinnän avulla *MainWindow*-luokka osaa yhdistää käyttöliittymän XAML:n sekä taustalla olevan koodin toisiinsa.

xmlns määrittelee ohjelman käyttöön nimiavaruuden. Pääoliossa täytyy aina määritellä vähintään yksi nimiavaruus, jota käytetään määrittämään itse pääolio ja kaikki sen lapsielementit. Jos ohjelman käyttöön halutaan määritellä useampi nimiavaruus, niin jokaiselle täytyy antaa uniikki etuliite, jota käytetään tunnistamaan kyseessä oleva nimiavaruus. Yleensä WPF-ohjelmat määrittelevät kaksi nimiavaruutta ohjelman käyttöön sen pääoliossa. Esimerkissä ensimmäisenä määritellään käyttöliittymäluokkien nimiavaruus vastaamaan .NET Frameworkin System.Windows.Controls-nimiavaruutta: *xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"*. Toisena määritellään tyyppien nimiavaruus vastaamaan System.Windows.Markup-nimiavaruutta, joka lisäksi määrittelee erityisiä toimintaohjeita XAML-kääntäjälle ja -jäsentäjälle: *xmlns:x=http://schemas.microsoft.com/winfx/2006/xaml*.

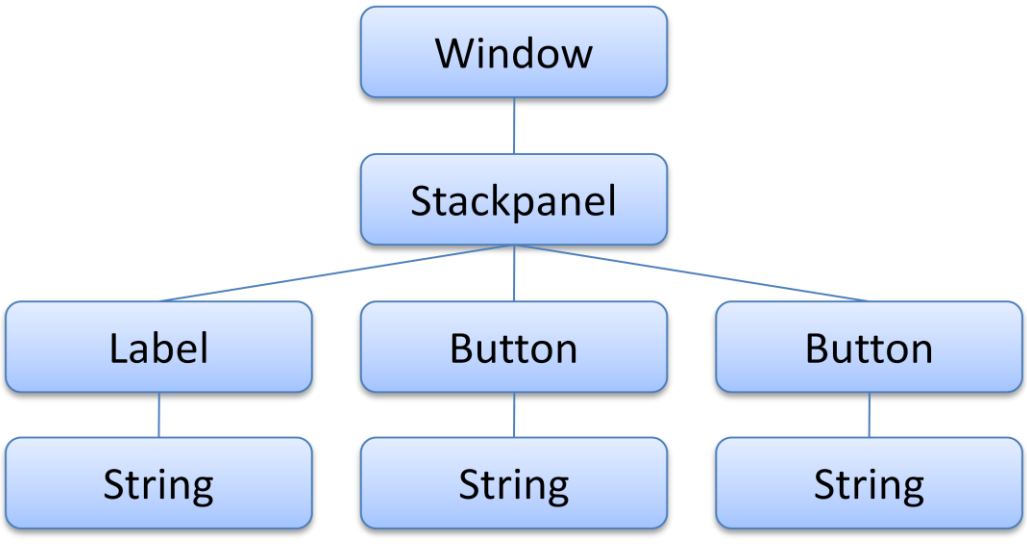
5.1.2 Looginen ja visuaalinen puu

XAML on luonteva esittämään käyttöliittymää sen hierarkkisen luonteensa takia. WPF:n käyttöliittymät rakennetaan olioiden puusta, joka tunnettaan myös nimel-

lä looginen puu. Looginen puu on olemassa käyttöliittymälle, vaikka käyttöliittymä ei olisi luotuna XAML:lla. Vaikka loogisen puun konsepti on hyvin yksinkertainen, niin se on erittäin tärkeä WPF-ohjelman toiminnan kannalta, koska melkein kaikki WPF:n toiminnallisuudet ovat yhdistettynä siihen (Nathan 2010, s. 75 - 76.). Esimerkiksi seuraavassa luvussa käytävät ominaisuudet ovat riippuvaisia loogisesta puusta.

Alla oleva esimerkki muuttaa XAML-luvun esimerkkiä hieman monimutkaisemmaksi, jolla saadaan näytettyä paremmin loogisen puun rakennetta. Kuva 5 esittää loogisen puun rakennetta alla olevalle esimerkille.

```
<Window x:Class="WPF_esimerkki.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        SizeToContent="WidthAndHeight">
    <StackPanel>
        <TextBlock Text="Looginen puu esimerkki"
                   Padding="10"/>
        <Button Content="Kyllä" />
        <Button Content="Ei" />
    </StackPanel>
</Window>
```



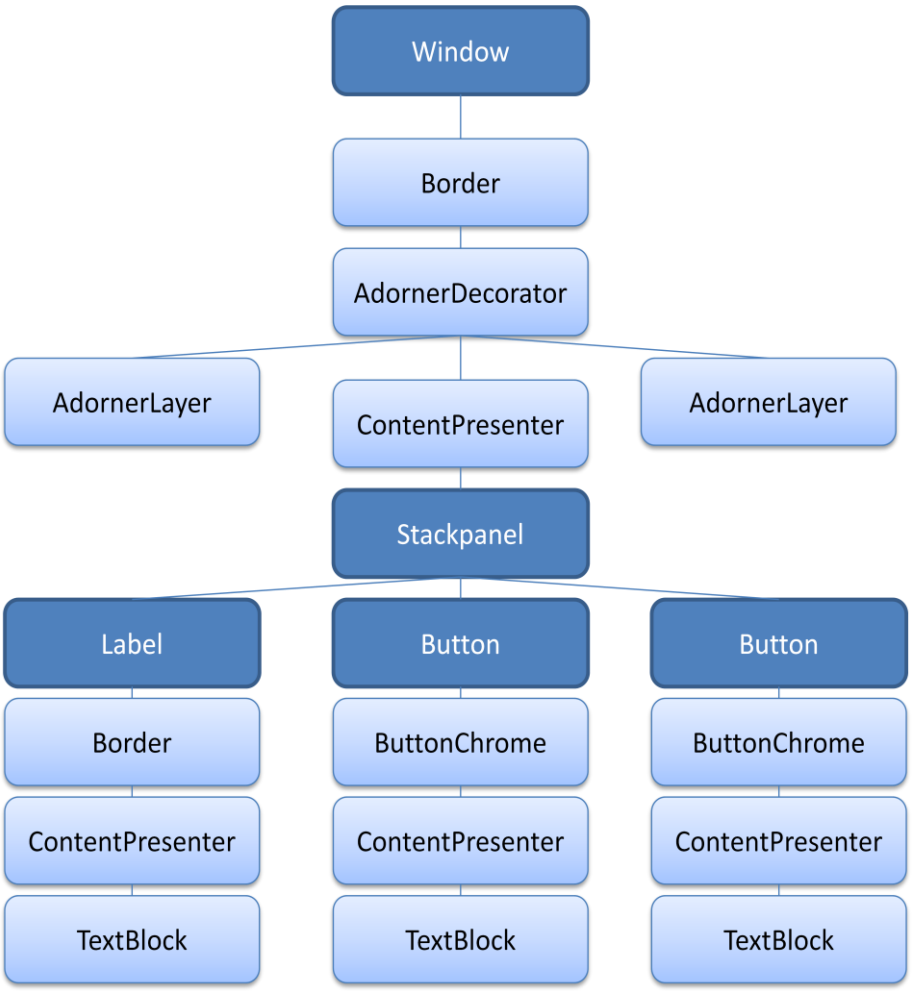
Kuva 5. Looginen puu esimerkkikäyttöliittymälle.

Loogisen puun tarkoitus on antaa sisällön malleille helppo tapa selata läpi niiden mahdollisia lapsiolioita sekä mahdollisuus laajentaa sisältömalleja uusilla

olioilla. Looginen puu tarjoaa myös kehyksen tietyille ilmoituksille, kuten milloin kaikki oliot loogisessa puussa ovat ladattu. Periaatteessa looginen puu on pääsääntöisesti ajonaikainen oliokaavio kehyksen tasolla, joka ei sisällä grafiikkaa, mutta on riittävä moniin kyselyoperaatioihin, jotka suoritetaan ajoaikaisesti ohjelman rakenteeseen. (Microsoft, Trees in WPF)

WPF:n looginen puu on pelkistys siitä, mitä oikeasti tapahtuu, kun elementtejä renderöidään. Visuaalista puuta sen sijaan voidaan ajatella loogisen puun laajennuksena, jossa kaikki solmut puretaan niiden visuaalisiksi peruskomponenteiksi. (Nathan 2010, s.75 – 76.)

Kuva 6 esittää aikaisemman esimerkin XAML-koodin visuaalisen puun. Huomioitavaa kuitenkin on, että kaikki loogisen puun solmut eivät esiinny visuaalisessa puussa. Pelkästään elementit, jotka periytyvät System.Windows.Media.Visual- tai System.Windows.Media.Visual3D-luokista sisältyvät visuaaliseen puuhun. Muut elementit eivät sisälly visuaaliseen puuhun, koska niillä ei ole omaa renderöinti ominaisuutta. Esimerkiksi yllä olevassa loogisen puun kuvassa oleva String ei sisälly visuaaliseen puuhun.



Kuva 6. Visuaalinen puu esimerkki käyttöliittymälle.

Koska visuaalinen puu sallii katsomisen WPF:n elementtien koostumukseen, ne voivat olla yllättävän monimutkaisia. Visuaaliset puut ovat olennainen osa WPF:n infrastruktuuria, mutta onneksi niistä ei tarvitse huolehtia, ellei huomattavasti muokkaa elementtien ulkoasua. Sellaisen ohjelman koodin kirjoittaminen, joka riippuu visuaalisen puun rakenteesta, rikkoo yhden WPF:n tärkeimmistä periaatteista, ohjelman käyttöliittymän ja toiminnallisuuden erottamisen toisistaan. (Nathan 2010, s. 77.)

5.1.3 Riippuvuusominaisuudet

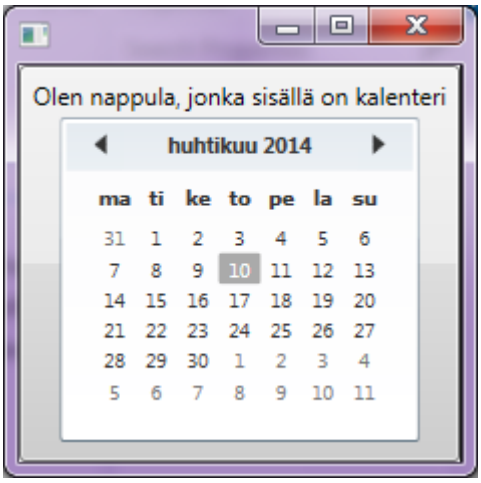
Aikaisemmassa esimerkissä (kuva 4) käytettiin myös luokkien omaisuuksia (property), joiden avulla pystytään yksinkertaisesti muokkaamaan esimerkiksi käyttöliittymäolion ulkomuotoa tai yleistä toimintaa. Window-luokalle määriteltiin

esimerkissä yksi ominaisuus, jonka avulla pystytään rajaamaan ikkunan koko vastaamaan sen sisällön kokoa, niin pysty- kuin leveyssuunnassa. *SizeToContent="WidthAndHeight"* on koko ominaisuuden määrittely, jossa SizeToContent on ominaisuuden nimi ja "*WidthAndHeight*" on ominaisuudelle määritelty arvo. Muita vaihtoehtoisia arvoja, joita *SizeToContent*ille voitaisiin määritellä, ovat muun muassa "*Height*" ja "*Manual*".

Myös Button-luokalle on määritelty yksi ominaisuus, Content, joka määrittelee näytettävän Button-luokan sisällön. Sisältö voi olla määriteltynä esimerkiksi tekstiksi, kuten esimerkissä tai toisen käyttöliittymän elementin sisällöksi.

Tässä piilee myös yksi WPF:n tehokkaimmista erityispiirteistä, valmiiden käyttöliittymäelementtien näkymän muokattavuus yksinkertaisesti XAML:n avulla. Tämä mahdollistaa ulkoasullisesti erittäin näyttävien ohjelmien kehittämisen hyvin pienellä vaivalla, mutta luo mahdollisesti ohjelman käyttäjille myös hieman sekalaisen käyttökokemuksen ellei käyttöliittymäsuunnittelija ole tehnyt järkeviä ratkaisuja. Seuraavassa esimerkissä Button-luokan sisällöksi on asetettu teksti ja tekstin alapuolelle Calendar-luokka, joka varmasti onnistuu hämmentämään ohjelman käyttäjää. Kuva 7 esittää alla olevan esimerkki XAML:n näkymän, joka on täysin toimiva Button-luokan muokkaus.

```
<Grid>
   <Button Padding="5">
      <Button.Content>
         <StackPanel>
            <TextBlock Text="Olen nappula, jonka sisällä on kalenteri"/>
            <Calendar/>
         </StackPanel>
      </Button.Content>
   </Button>
</Grid>
```



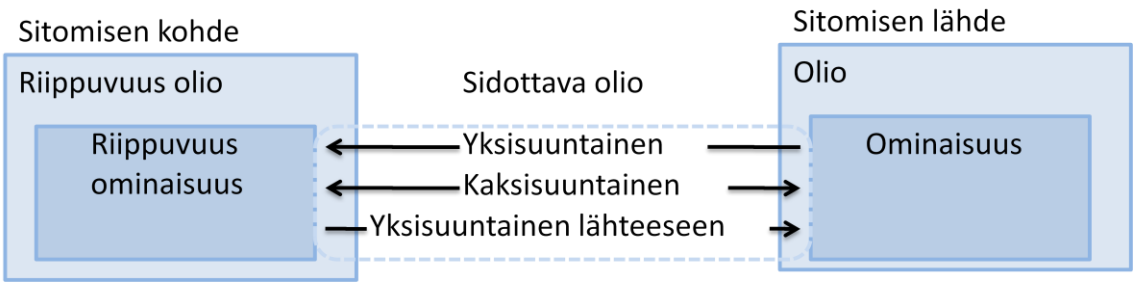
Kuva 7. WPF käyttöliittymäelementtien muokkausmahdollisuus.

5.1.4 Tiedonsidonta

WPF:n tiedonsidonta tarjoaa ohjelmille yksinkertaisen ja johdonmukaisen tavan esittää ja olla vuorovaikutuksessa tiedon kanssa. Käyttöliittymän elementit voidaan sitoa tietoon laajasta tietolähteiden joukosta, CLR-olioiden ja XML:n muodossa. Tiedonsidonnan toiminnallisuus WPF:ssä tarjoaa lukuisia etuja tavallisiin malleihin verrattuna, sisältäen laajan valikoiman ominaisuuksia, jotka luonnostaan tukevat tiedonsidontaa, joustavan käyttöliittymän tiedon esittämiseen ja menetelmän puhtaasti erottaa ohjelman toimintalogiikan käyttöliittymästä. (Microsoft, Data Binding.)

Tiedonsidonta on prosessi, joka yhdistää käyttöliittymän elementit ohjelman toimintalogiikkaan. Tiedonsidonnan avulla käyttöliittymässä tehtävät muutokset, esimerkiksi tekstikenttään kirjoitettava teksti vaikuttavat myös taustalla oleviin ominaisuuksiin, jos tiedonsidonta on tehty oikealla tavalla. Tämän ansiosta ohjelman taustalla oleva koodi voi tehdä esitettävään tietoon muokkauksia, esimerkiksi järjestellä tietoa tai suodattaa tietoa tarpeen mukaan pois, jolloin käyttöliittymä näyttää pelkästään muokatut tiedot.

Tiedonsidonnassa on useita eri asetusvaihtoehtoja, jotka määrittelevät, kuinka tiedonsidonta tulee toimimaan käyttöliittymäelementin ja ominaisuuden välillä. WPF tarjoaa viisi erilaista tilaa tiedonsidontaan, jotka ovat oletus, yhden kerran, yksisuuntainen, kaksisuuntainen ja yksisuuntainen lähteeseen. Kuva 8 esittää kolmen yleisimmin käytetyn tiedonsidonnan tietojen virran.



Kuva 8. Tiedonsidonnan riippuvuudet lähteen ja kohteen välillä.

Yksisuuntaisella sidonnalla tarkoitetaan, että vain sidonnan lähteen ominaisuuteen tehdyt muutokset tulevat vaikuttamaan lähteen ominaisuuteen, vaikka käyttöliittymä mahdollistaisi riippuvuusominaisuuden muokkaamisen esimerkiksi tekstikentässä. Yksisuuntaista sitomista käytetään yleensä toteuttamaan käyttöliittymän elementille pelkästään lukemisoikeus.

Kaksisuuntaisella sidonnalla tarkoitetaan, että sidonnan lähteeseen ja kohteeseen tehdyt muutokset vaikuttavat aina myös sidonnan lähteen ominaisuuteen. Kaksisuuntaista sitomista käytetään toteuttamaan käyttöliittymän elementeille luku- ja kirjoitusoikeudet. Koska kaksisuuntaisessa sitomisessa joudutaan aina monitoroimaan myös riippuvuusominaisuutta, sidonta aiheuttaa suuremman kustannuksen kuin yksisuuntaisen sidonta-tilan käyttäminen (Microsoft, Data Binding).

Yksisuuntainen lähteeseen sidonta on käänteinen verrattuna yksisuuntaiseen sidontaan, jolloin muutokset käyttöliittymän elementin riippuvuusominaisuuteen päivittyvät suoraan sitomisen lähteen ominaisuuden. Esimerkkitapaus, jossa yksisuuntaista lähteeseen sidontaa voi hyödyntää, on uudelleen arvioida sitomisen lähteen ominaisuuden arvoa käyttöliittymän elementissä.

5.1.5 Säikeistysmalli

WPF on suunniteltu säästämään ohjelmoijat säikeistyksen ongelmilta. Tämän ansiosta suurimman osan kehittäjistä ei tarvitse kirjoittaa käyttöliittymää, joka käyttää useampaa kuin yhtä säiettä. Monisäikeiset ohjelmat ovat monimutkaisia ja ne ovat vaikeita tarkistaa virheiden osalta, jonka takia monisäikeisiä ratkaisuja tulisi aina välttää, jos ohjelma on myös mahdollista toteuttaa yksisäikeisenä. (Microsoft, Threading Model.)

Tyypillisesti WPF-ohjelmat käynnistyvät kahdella säikeellä. Yksi säie hoitaa renderöinnin ja toinen säie hoitaa käyttöliittymän. Renderöintisäie suorittaa tehtäviään tehokkaasti piilotettuna ohjelman taustalla, kun käyttöliittymän säie muun muassa vastaanottaa ohjelman käyttäjän syötteitä, hoitaa tapahtumia ja suorittaa ohjelman koodia. Suurin osa ohjelmista käyttää vain yhtä säiettä käyttöliittymään, vaikka joissain tilanteissa useamman säikeen samanaikainen käyttö on suositeltavaa. (Microsoft, Threading Model.)

Käyttöliittymän säie laittaa sille tulevat tehtävät jonoon Dispatcher-olion sisään, joka valitsee jonossa olevia tehtäviä suoritukseen sen tärkeysasteen mukaan. Jokaisella käyttöliittymän säikeellä on oltava vähintään yksi Dispatcher-olio, jonka lisäksi jokainen Dispatcher-olio voi suorittaa tehtäviä vain yhdessä säikeessä. (Microsoft, Threading Model.)

Tämä säikeistysmalli voi estää käyttöliittymää vastaamasta ohjelman käyttäjälle, jos esimerkiksi ohjelma hakee tietokannasta kerralla paljon tietoa ja tämä tehtävä suoritetaan käyttöliittymän säikeessä. Tehtävän suorituksen aikana Dispatcher ei voi vastaan ottaa uusia tehtäviä tai suorittaa samanaikaisesti muita tehtäviä, jonka takia käyttöliittymän säikeessä suoritettavat tehtävät tulisi pitää mahdollisimman lyhyinä. Jos käyttöliittymän säikeessä on tehtävä, joka suorittaa itseään aina kymmenen sekunnin ajan, niin myös käyttöliittymän toiminta pysähtyy aina kymmenen sekunnin ajaksi.

Ratkaisu tähän ongelmaan on suorittaa pitkäkestoiset tehtävät erillisessä säikeessä. Tämä mahdollistaa, että käyttöliittymä tulee vastaamaan käyttäjälle, vaikka pitkäkestoinen tehtävä olisikin kyseisellä hetkellä suorituksessa. Mikäli tehtävät, jotka suoritetaan muussa kuin käyttöliittymän säikeessä vaikuttavat käyttöliittymään (esimerkiksi esitettäviin tietoihin), pitää käyttöliittymään vaikuttava koodin suoritus siirtää Dispatcher-oliolle. Dispatcher siirtää koodin suorituksen käyttöliittymäsäikeeseen ja mahdollistaa käyttöliittymän tietojen muuttamisen.

5.2 Entity Framework

Entity Framework (EF) on Microsoftin kehittämä teknologia, joka yhtenäistää relaatiotietokantakyselyiden syntaksin. EF on joukko teknologioita, jotka tukevat

tietoon suuntautuneiden ohjelmien kehittämistä. EF sallii kehittäjien työskennellä aluekohtaisten luokkien ja ominaisuuksien kanssa ilman, että kehittäjien tarvitsee huolehtia alla olevien tietokantojen tauluista ja sarakkeista, joihin tieto on tallennettu. Näin kehittäjät pystyvät toimimaan korkeammalla abstraktion tasolla, jolloin esimerkiksi tietoon suuntautuneiden ohjelmien luominen ja ylläpito vaatii vähemmän koodia kuin perinteiset ohjelmat. (Microsoft, Entity Framework.)

EF on ORM-kartoittaja (Object-relational mapper, ORM), joka vähentää epäsopivuuksia .NET Frameworkin olio-ohjelmoinnin ja relaatiotietokantojen välillä. EF sallii kehittäjien pääsäännöllisesti olla yhteydessä ohjelman käsitteelliseen malliin käyttäen tuttuja oliosuuntautuneita tekniikoita. Kehittäjät voivat suorittaa tietokantaoperaatioita ohjelman käsitteelliseen malliin, jolloin EF kääntää operaatiot tietokantatoimintoihin. (Microsoft, Entity Framework.)

Ensimmäinen julkinen versio EF:stä julkaistiin vuonna 2008 Visual Studio 2008 Service Pack 1:n ja .NET Service Pack 1:n yhteydessä. Tämä tapahtui kaksi vuotta sen jälkeen, kun Microsoft ensimmäisen kerran julkisti sen TechEd 2006-konferenssissa. Ensimmäinen versio EF:sä otettiin vastaan epäilevästi ja sen toteutus oli kaukana toimivasta. Kaksi vuotta myöhemmin Visual Studio 2010 ja .NET 4:n julkaisemisen yhteydessä EF oli kehittynyt huomattavasti eteenpäin ja alkoi saamaan kasvavaa huomiota ja kiinnostusta ohjelmistokehittäjiltä. EF:stä onkin alkanut muodostumaan Microsoftin ydin tiedonsaantialusta .NET sovellusten kehittämiseen. (Lerman 2010, s. 1.)

5.3 .NET Framework

.NET Framework koostuu Common Language Runtime:sta (CLR) ja laajasta sekä monipuolisesta luokkakirjastosta. CLR on .NET Framework:n perusta, joka hallitsee ajoaikana suoritettavaa koodia tarjoten samalla ydinominaisuuksia, kuten esimerkiksi muistin- ja säiehallintaa. Tämän lisäksi CLR myös valvoo tyyppiturvallisuutta ja muita koodin tarkkuuden muotoja, jotka korostavat turvallisuutta. (Microsoft, .NET Framework.)

Luokkakirjasto on kattava kokoelma olio-ohjelmointiin suuntautuneita ja uudelleen käytettäviä luokkia, joita käyttää todella monipuoliseen ohjelmiston kehityk-

seen aina perinteisestä konsolipohjaisesta ohjelmasta graafisien käyttöliittymien toteuttamiseen (Microsoft, .NET Framework). Näiden valmiiden luokkien käyttäminen mahdollistaa erittäin nopean ja tehokkaan kehitysaikataulun, koska jokaiseen uuteen asiaan ei tarvitse itse kehittää luokkaa, vaan pystyy hyödyntämään Microsoftin jo etukäteen hyvin suunniteltuja ja testattuja luokkia. Tämän ansiosta ohjelmistokehittäjien on mahdollista keskittyä enemmän ohjelman toimintalogiikkaan, eikä uudelleen keksimään jo valmiita toimivia ohjelmakomponentteja, kuten esimerkiksi ajastimia.

6 C#-ohjelmointikieli

C# on Microsoftin luoma yleiskäyttöinen olio-ohjelmointikieli, jonka päätavoitteena on ohjelmoijan tuottavuus. Tämän takia C# on kirjoitettu toimimaan erittäin hyvin Microsoftin .NET Framework:n kanssa, jonka avulla ohjelmoija pystyy käyttämään valmiita ja hyvin testattuja komponentteja. C# tasapainottelee yksinkertaisuuden, ilmaisukyvyn ja suorituskyvyn välillä. (Nathan, 2010.)

6.1 Historia

1980-luvulla suurin osa ohjelmista, jotka kirjoitettiin toimimaan Windows-käyttöliittymällä, toteutettiin C++-ohjelmointikielellä. Vaikka C++ on olio-ohjelmointikieli, niin sen hallitseminen on kiistatta hyvin vaikeaa. Ohjelmoija on C++-kieltä käyttäessään itse vastuussa muistinhallinnasta ja tietoturvallisuusasioita. (Clark 2011, s. 5.)

1990-luvulla Java-ohjelmointikielestä tuli suosittu. Java on hallittu ohjelmointikieli tarkoittaen, että se vapauttaa ohjelmoijan esimerkiksi muistinhallintaan liittyvistä ongelmista. Hallitut ohjelmointikielet tarjoavat tavan käsitellä esimerkiksi muistinhallintaa ja roskien keräämistä, jolloin ohjelmoijalle jää enemmän aikaa keskittyä itse ohjelman toimintalogiikkaan. (Clark 2011, s. 6.)

Kun Microsoft huomasi Javan läpilyönnin ja Internetin kasvaneen suosion, niin se päätti kehittää oman joukon hallittuja ohjelmointikieliä. Microsoft halusi, että kehittäjien on helppo kehittää niin työpöytäsovelluksia Windows-käyttöliittymälle kuin selainpohjaisia sovelluksia. Nämä hallitut kielet, Visual Basic ja C#, perus-

tavat toimintansa .NET Frameworkiin, joka tarjoaa niille suuren osan toiminnallisuudesta, jota vaaditaan kaikissa ohjelmissa. (Clark 2011, s. 6.)

.NET Frameworkin kehityksen aikana sen luokkakirjastot kirjoitettiin uudella ohjelmointikielellä, C#:lla. C#:n pääsuunnittelija sekä johtava arkkitehti on tanskalainen Anders Hejlsberg, joka on aikaisemmin ollut suunnittelemassa Turbo Pascal- ja Delphi-ohjelmointikieliä. Heljsberg käytti aikaisempaa kokemustaan hyväkseen suunnitellessaan uutta olio-ohjelmointikieltä, joka rakennettiin näiden ohjelmointikielien vahvuuksien päälle sekä korjaten niiden heikkouksia. Anders Heljsberg suunnitteli C#:n syntaksin pitkälti C-ohjelmointikielen pohjalta miellyttääkseen C++- ja Java-kehittäjiä. (Clark 2011, s. 6.)

C# julkaistiin yleiseen beetatestaukseen kesäkuussa 2000. Ohjelmointikielen virallinen julkistaminen tapahtui kaksi vuotta myöhemmin, keväällä 2002, jolloin sen ensimmäinen versio julkaistiin. (Jones 2004, s. 7.)

6.2 Tietotyypit

Tietotyyppi määrittelee toimintasuunnitelman arvolle (Albahari & Albahari 2012, s. 15). C# on vahvasti tyypitetty ohjelmointikieli, jossa jokaisella muuttujalla on oma tyyppinsä ja jossa jokainen lauseke arvioidaan arvoon. Tietotyyppiin varastoidut tiedot voivat sisältää muun muassa:

- varastointitilan, jonka tietotyyppi tarvitsee
- maksimi- ja minimiarvot, joita tietotyyppi voi esittää
- jäsenet, joita se sisältää (esimerkiksi metodit ja kentät)
- kantaluokan, josta se on periytetty. (Microsoft, Types.)

C# tarjoaa standardin joukon sisäänrakennettuja tietotyyppejä, jotka esittävät kokonaislukuja, desimaaleja, tosi/epätosi arviot, tekstimerkkejä ja muita tiedon tyyppejä. Näiden lisäksi C#-ohjelmointikieleen on myös sisäänrakennettu string- ja object-tyypit, joista string-tyyppi kuvaa merkkijonoja ja object-tyyppiä käytetään kaikkien luokkien kantatyyppinä. (Microsoft, Types.). Kaikki C#:n tietotyypit kuuluvat seuraaviin kategorioihin: arvotyypit, viittaustyypit, yleistyyppiparametrit ja pointterityypit (Albahari & Albahari 2012, s. 19.).

Seuraavissa aliluvuissa käsitellään vain C#-ohjelmoinnissa yleisimmin käytetyt arvo- ja viittaustyypit.

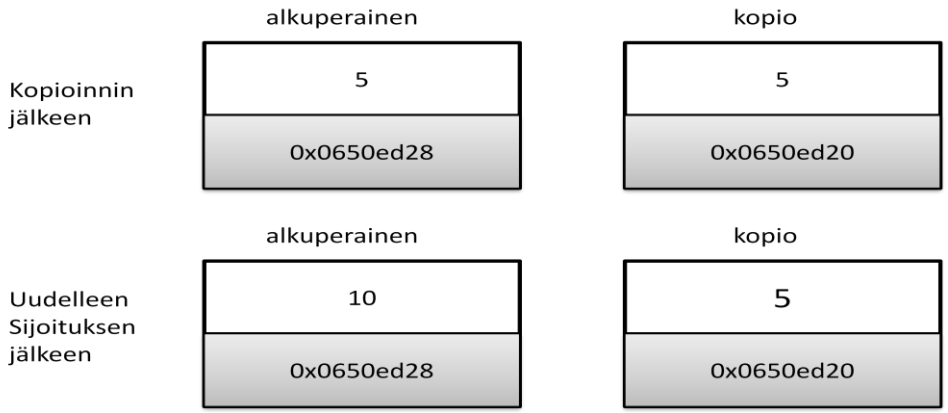
6.2.1 Arvotyypit

Muuttujat, jotka perustuvat arvotyyppeihin, sisältävät suoraan arvoja. Arvotyyppimuuttujan sijoittaminen toiseen arvotyyppimuuttujaan kopioi arvon, joka eroaa viittaustyyppimuuttujien sijoittamisesta, jossa viittaus muuttujaan kopioidaan, eikä muuttujaa itsessään. Kaikki arvotyypit periytyvät System.ValueType-luokasta, joka taas periytyy System.Object-luokasta. (Microsoft, Value Types.)

Arvotyypit eroavat viittaustyypeistä erityisesti kahdessa asiassa. Arvotyyppiä ei voi käyttää kantaluokkana, eli siitä ei voi periyttää uusia tietotyyppejä eikä arvotyyppi voi sisältää tyhjää (null) arvoa. Tosin arvotyypeille lisättiin C# 2.0-versiossa tyhjätyyppi-ominaisuus (nullable types), johon pystyy sijoittamaan myös tyhjän arvon. (Microsoft, Value Types.)

Alla olevassa koodissa ensin alustetaan int-kokonaisluku, *alkuperainen*, joka on C#-kieleen sisäänrakennettu arvotyyppi. Sitten tämä sijoitetaan toiseen int-kokonaislukuun, jolloin arvo kopioidaan uuteen muuttujaan. *alkuperainen-muuttujaan* sijoitetaan uusi luku kokonaisluku 10, joka ei tule muuttamaan kopion arvoa. Kuva 9 näyttää, että *alkuperainen*- ja *kopio*-muuttujilla on omat säilytystilansa kopioinnin ja uudelleen sijoittamisen jälkeen. Kuvan laatikoissa yläpuolella oleva luku kuvaa muuttujan arvoa ja alapuolella oleva merkkijono kuvaa muistiosoitetta, jossa muuttujaa säilytetään.

```
int alkuperainen = 5;
int kopio = alkuperainen;
alkuperainen = 10;
```

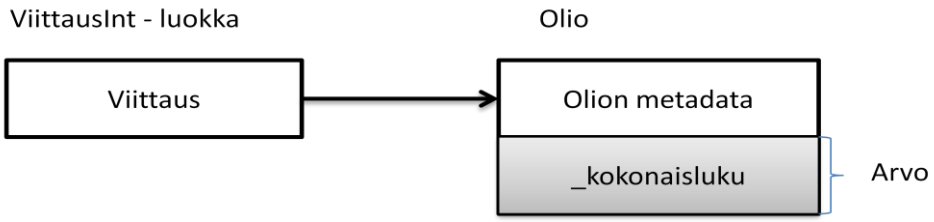


Kuva 9. Sijoittaminen kopioi vain arvon.

6.2.2 Viittaustyypit

Viittaustyypit ovat monimutkaisempia kuin arvotyypit, koska ne koostuvat kahdesta eri osasta, oliosta ja viittauksesta kyseiseen olioon. Toisin sanoen viittaustyypit itsessään eivät suoraan sisällä niiden tietoja. Viittaustyypin sisältö on vain viittaus olioon, joka sisältää oikeat arvot. (Albahari & Albahari 2012, s. 20). Kuva 10 näyttää viittaustyypin sijoittumisen ohjelman muistiin, jonka viittaustyyppinä on käytetty seuraavanlaista luokkaa:

class ViittausInt { public int _kokonaisluku; }

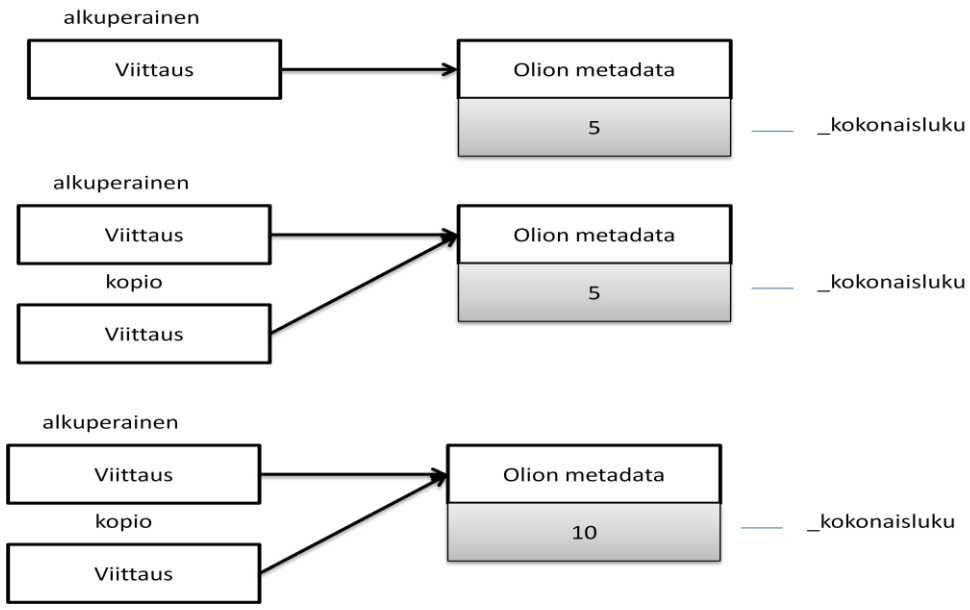


Kuva 10. Viittaustyypin ilmentymä muistissa.

Tämä vaikuttaa viittaustyyppien kopioimiseen, jolloin sijoitusoperaatiossa kopioidaan pelkästään muistiosoite eikä itse arvoja. Asiaa selventämään muunnetaan Arvotyypit-luvun esimerkki vastaamaan samanlaista sijoitusoperaatioita viittaustyypille. Kuva 11 näyttää sijoitusoperaatioiden lopputuloksen. Aluksi luodaan uusi luokan ilmentymä, jonka kokonaisluvun arvoksi sijoitetaan 5. Seuraavaksi *alkuperainen*-muuttuja sijoitetaan kopio-muuttujaan, jonka seurauksesta *alkuperainen*-muuttujan viittauksen muistiosoite sijoitetaan *kopio-muuttujaan.* Eli

tämän jälkeen molemmat muuttujat viittaavat samaan muistiosoitteeseen, jolloin toisen muuttujan *_kokonaisluku*-jäsenmuuttujan muokkaaminen vaikuttaa vastaavasti toiseen muuttujaan.

```
ViittausInt alkuperainen = new ViittausInt() { _kokonaisluku = 5 };
ViittausInt kopio = alkuperainen;
alkuperainen._kokonaisluku = 10;
```



Kuva 11. Vittaustyypin sijoittaminen aiheuttaa viittauksen sijoittamisen.

6.3 Muistinhallinta

C#-kieli luottaa, että Common Language Runtime (CLR) suorittaa automaattisen muistinhallinnan. CLR:ssä on roskienkeruuominaisuus, joka suoritetaan osana ohjelmaa keräten olioiden muistia, joihin ei ole enää viittauksia. Automaattisen muistinhallinnan ansiosta ohjelmoijan ei tarvitse huolehtia muistin vapauttamisesta, jonka ansiosta voidaan välttää väärien pointterien ongelma, jota kohdataan esimerkiksi C++-ohjelmointikielessä. (Albahari & Albahari 2012, s. 2-3.)

Pointtereita ei kuitenkaan ole poistettu C#-kielestä, vaan niitä pidetään turhina useimpiin ohjelmointitehtäviin. Suorituskykykriittisissä ja yhteensopivuutta tilanteissa pointtereita voidaan kuitenkin käyttää. Niiden käyttäminen tosin vaatii

erikseen koodilohkon merkitsemisen unsafe-avainsanalla. (Albahari & Albahari 2012, s. 3.)

6.4 LINQ

Language-Intergrated Query (LINQ) on joukko ominaisuuksia, jotka esiteltiin Visual Studio 2008 -version yhteydessä. LINQ laajentaa tehokkaita kyselymahdollisuuksia C#- ja Visual Basic-ohjelmointikielien syntakseihin. LINQ sisältää helposti opittavissa olevia kaavoja kyselyiden ja tietojen päivittämisen tekemiseen sekä teknologian voi käytännössä laajentaa tukemaan mitä tahansa tietovarastoa. Visual Studio sisältää mahdollisuuden käyttää LINQ:ta muun muassa .NET Frameworkin kokoelmien (collection) ja XML dokumenttien kanssa. (Microsoft, LINQ.)

LINQ tarjoaa mahdollisuuden kirjoittaa jäsenneltyjä tyyppiturvallisia kyselyitä paikallisiin tai etäisiin tietovarastoihin. Teknologia esiteltiin ensimmäisen kerran osana C#-ohjelmointikieltä sen versiossa 3.0 ja .NET Frameworkin versiossa 3.5. (Albahari & Albahari 2012, s. 319.)

Yksi tärkeimmistä LINQ:n ominaisuuksista on sen tarjoama yhtenäinen syntaksi eri tietovarastojen välillä. Siinä missä eri tietokantojen SQL-syntaksit saattavat erota toisistaan, LINQ-teknologia mahdollistaa tietokantojen vaihtamisen kesken ohjelman kehityksen ilman tarvetta muuttaa ohjelmassa olevia tietokantakyselyiden lauseita. Esimerkiksi MySQL-tietokantakysely eroaa syntaksinsa puolesta SQL Server-tietokantakyselyn syntaksista, kun haetaan tietoa ja palautettava tiedonmäärä rajataan.

MySQL-tietokantakyselyn syntaksi:
*SELECT * FROM taulu LIMIT kpl;*

SQL Server-tietokantakyselyn syntaksi:
SELECT TOP kpl FROM taulu;

Mikäli ohjelman tietokantayhteyksiin on käytetty Microsoftin EF:ää, niin yllä olevan esimerkin tapaista ongelmaa ei ohjelmassa tule esiintymään, vaikka MySQL-tietokannan vaihtaisi SQL Server-tietokantaan. Sama periaate toimii

myös .NET Frameworkin kokoelmien välillä, mikä mahdollistaa tietorakenteen vaivattoman vaihtamisen ohjelman kehityksessä.

LINQ tarjoaa kaksi erilaista syntaksia ohjelmoijien käyttöön, jotka ovat nimeltään method syntax ja query syntax. Syntaksit ovat toimintansa puolesta identtiset, mutta monien ihmisten mielestä query syntax on yksinkertaisempi ja helpompi lukea. (Microsoft, LINQ Syntax.)

Esimerkki query syntaxista ja method syntaxista, joille tehdään sama kysely kuin aikaisemmassa MySQL- ja SQL Server-tietokantakyselyissä:

Query syntax:
(from tietue in taulu select tietue).Take(kpl);

Method syntax:
taulu.Take(kpl);

6.5 Attribuutit

Attribuutit ovat laajennettavuusmekanismi, jolla pystytään lisäämään itse määriteltyjä tietoja koodielementteihin, kuten esimerkiksi tyyppeihin, palautusarvoihin ja parametreihin. Laajennettavuusmekanismi on hyödyllinen ominaisuus, joka yhdentyy syvästi tyyppisysteemiin ilman, että jouduttaisiin luomaan uusia avainsanoja tai rakenteita ohjelmointikieleen. (Albahari & Albahari 2012, s. 173.)

Attribuuteilla on seuraavia ominaisuuksia (Microsoft, Attributes):

- Voidaan lisätä metadata ohjelmaan.
- Voidaan lisätä yksi tai useampi attribuutti esimerkiksi luokille ja ominaisuuksille.
- Ne hyväksyvät samalla tavalla parametreja kuin metodit ja ominaisuudet.
- Voidaan tutkia ohjelman omaa tai toisten ohjelmien metadataa käyttämällä heijastusta (reflection).

Sarjallistaminen (serialization) on hyvä esimerkki attribuutin toiminnasta. Se on prosessi, jossa olio muunnetaan tavuvirraksi, jotta se voidaan tallentaa tai lähettää muistiin, tietokantaan tai tiedostoon. Sarjallistamisen päätarkoitus on tallen-

taa olion sen hetkinen tila, että se voidaan myöhemmin luoda uudestaan samassa tilassa, kun sitä tarvitaan. (Microsoft, Serialization.)

7 Visual Studio

Visual Studio on ohjelmistokehitysympäristö, jolla voi luoda muun muassa työpöytä-, mobiili- tai ASP.NET-sovelluksia. Visual Studiossa voidaan käyttää monia eri ohjelmointikieliä, kuten esimerkiksi Visual C#, Visual C++ tai Visual Basic. Visual Studion avulla voidaan luoda entistä helpommin ohjelmia, jotka käyttävät useampaa kuin yhtä ohjelmointikieltä. (Microsoft, Visual Studio.)

Visual Studio on suunniteltu tukemaan ohjelmoijien tuottavuutta ohjelman koodin kirjoittamisessa. Tuottavuuden lisäämiseksi Visual Studioon on lisätty ominaisuuksia, joita useat ohjelmoijat tarvitsevat jokapäiväisessä työssään. Seuraavissa alaluvuissa käydään läpi kaksi tärkeää ominaisuutta, IntelliSense ja Code Snippets.

7.1 IntelliSense

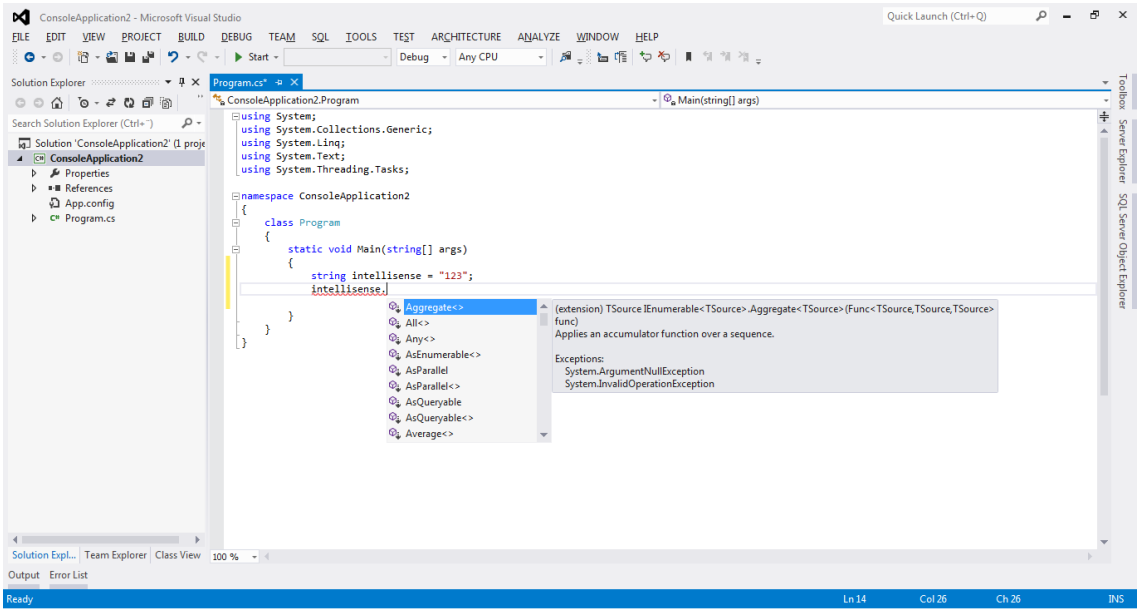
IntelliSense on yleinen termi useille eri ominaisuuksille: listan jäsenet (List Members), parametrin informaatio (Parameter Info), nopea informaatio (Quick Info) ja sanan täydennys (Complete Word). Näiden ominaisuuksien avulla pystyy oppimaan enemmän käytetystä koodista, seurata kirjoitettavia parametreja metodin kutsumisessa ja lisätä kutsuja ominaisuuksiin tai metodeihin vain muutamalla napin painalluksella. (Microsoft, IntelliSense.)

Aikaiset IntelliSense-versiot vaativat ensin määrittelemään luokan tai ominaisuuden nimen, että IntelliSense-ominaisuutta pystyttiin hyödyntämään. Nyt IntelliSense-ominaisuus aktivoituu heti, kun käyttäjä kirjoittaa ensimmäisen kirjaimen. Tämän ansiosta ohjelmoija pystyy nopeasti tunnistamaan muun muassa luokkia, komentoja sekä avainsanoja. (Sheldon & Hollis & Windsor & McCarter & Hillar & Herman. 2012, s. 35.)

Esimerkiksi luokan valinnan jälkeen IntelliSense sallii ohjelmoijan valita luokkaan kuuluvan metodin tai ominaisuuden. IntelliSense auttaa myös metodin

kutsumisessa näyttämällä listan kaikista metodin parametreista, joita sen kutsumiseen tarvitaan. (Sheldon ym. 2012, s. 35.)

IntelliSense-ominaisuuden toiminta perustuu kontekstiin, jossa koodia kirjoitetaan. Esimerkiksi luokan metodia kirjoittaessa IntelliSense tarjoaa myös mahdollisuuden käyttää luokan sisäisiä muuttujia, jotka eivät näy luokan ulkopuolelle eivätkä näin ollen näy myöskään IntelliSense:ssä luokan ulkopuolella. Toisin sanoen IntelliSense osaa poistaa kaikki epäsopivat kohdat listauksesta, joita ei kirjoituskohdassa voi käyttää. Kuva 12 näyttää IntelliSensen toiminnan Visual Studiossa. (Sheldon ym. 2012, s. 35.)



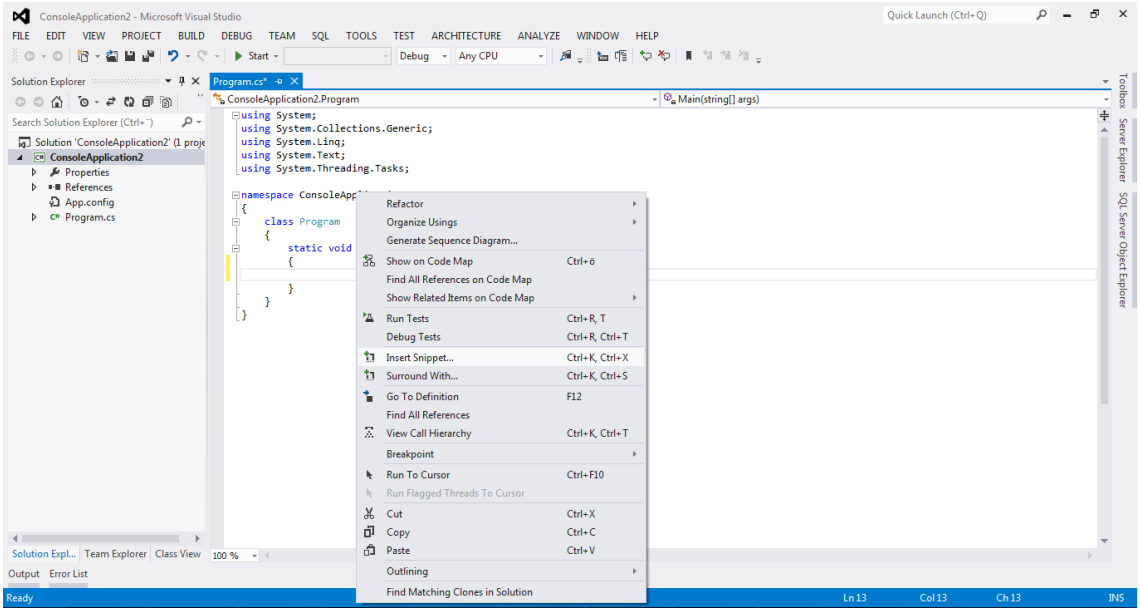
Kuva 12. IntelliSensen toiminta Visual Studiossa.

7.2 Code Snippets

Code Snippets -ominaisuus mahdollistaa uudelleen käyttää koodinpätkiä, jotka voidaan helposti lisätä ohjelman koodiin. Nämä koodinpätkät voivat olla yksinkertaisia tai monimutkaisia, joita voidaan käyttää yksinkertaisiin try-catch-lohkoihin, jossa tarkistetaan tapahtuuko ohjelman suorituksessa virhettä tai myös kokonaisten luokkien lisäämiseen valittuun koodin kohtaan. (Microsoft, Code Snippets.)

Koodinpätkien käyttö ei tapahdu samalla tavalla avainsanojen kautta kuin IntelliSensessä. Kooditiedostossa siirrytään siihen kohtaan, johon halutaan lisätä

valmis koodinpätkä ja painetaan hiiren oikealla näppäimellä, jonka jälkeen valikosta voidaan valita Insert Snippet…. Kuva 13 näyttää Code Snippets -valikon avaamisen.



Kuva 13. Valmistellaan koodinpätkän lisäämistä koodiin.

Visual Studion mukana tulee valmis kirjasto koodinpätkiä, joiden avulla voi hieman nopeuttaa useasti käytettävän koodin käyttöä. Code Snippets on valmiin kirjaston lisäksi täysin laajennettavissa, jonka avulla ohjelmoijat pystyvät kasvattamaan omaa tuottavuuttaan lisäämällä heidän useasti käyttämiään koodinpätkiä.

8 Opinnäytetyön vaiheet

Tässä luvussa käydään läpi opinnäytetyön eri vaiheet. Projekti suunniteltiin ja toteutettiin ketterää ohjelmistokehitystyyliä noudatellen, vaikka ei käytetty mitään tiettyä ketterää menetelmää, vaan muokattiin kehitysmenetelmä kyseiselle projektille parhaiten sopivaksi.

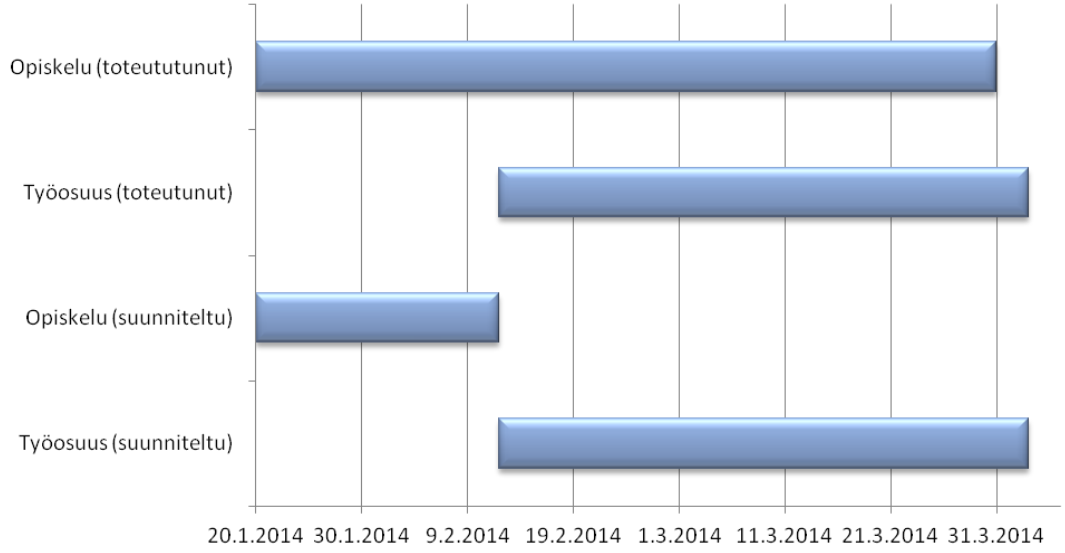
8.1 Projektin aikataulu

Projekti oli jaettu useampaan eri kokonaisuuteen, jotka kuitenkin menivät hyvin useasti päällekkäin. Päällekkäisyyden vuoksi aikataulu voitiin jakaa selvästi kahteen eri kokonaisuuteen: opiskeluun ja ohjelman toteutukseen.

Opinnäytetyön ensimmäisessä asiakaspalaverissa sovimme opinnäytetyössä käytettävistä tekniikoista ja teknologioista, joita opinnäytetyö tulisi vaatimaan. Ohjelmointikielenä FinFX Trading Oy:n toiveena oli käyttää Microsoftin C#-ohjelmointikieltä, koska se on heidän yrityksessään laajasti käytössä. Tietokantayhteyksiin valittiin Microsoftin Entity Framework, joka yhdistää useiden eri tietokantojen syntaksit, jolloin mahdollinen tietokannan vaihtaminen tulevaisuudessa ei aiheuta ohjelman kannalta mitään ongelmia. Käyttöliittymän toteutuksen toiveena oli käyttää Microsoftin Windows Presentation Foundation, joka tarjoaa tulevaisuuden kannalta parhaan kehitysalustan.

Kaikki opiskeltavat asiat ovat laajuudeltaan erittäin suuria, jonka takia varasin aikaa näiden asioiden opiskeluun noin sata tuntia. Asioiden opiskelun jälkeen ja itse opinnäytetyön tekemisen aloittaminen painottui vielä suuresti näiden asioiden käytön opettelemiseen. Kokonaisuutena opiskeltavaa oli huomattavasti enemmän kuin olin alun perin ajatellut, koska opinnäytetyön edetessä tuli useamman kerran uusia opiskeltavia asioita eteen.

Ohjelman toteutus piti sisällään ohjelman arkkitehtuurin suunnittelun, toteutuksen, jatkuvan testaamisen ja melkein päivittäiset lyhyet palaverit asiakkaan kanssa. Toteutukseen varattu aikataulu piti paikkansa erittäin hyvin, vaikka jouduin opiskelemaan melko paljon vielä toteutusvaiheessa. Kuvassa 14 esitellään arvio opinnäytetyön ja lopulta toteutunut määrä.



Kuva 14. Opinnäytetyön toteutunut aikataulu.

8.2 Opiskelu

Opinnäytetyöhön liittyvien tekniikoiden opiskelu käynnistyi heti ensimmäisen palaverin jälkeen, jolloin selvisi, millä tekniikoilla FinFX Trading Oy haluaisi ohjelman toteutettavan. Opiskelu alkoi jokaiseen aihealueeseen sopivan kirjan valinnalla, jotka pystyisivät kohtuullisen nopeasti opettamaan kyseisen teknologian periaatteet ja käyttämisen.

C#-ohjelmointikielen opiskeluun en varannut suuresti aikaa, koska olio-ohjelmointi oli minulle jo tuttua C++-ohjelmointikielen kokemukseni kautta. Tämän ansiosta pystyin nopeasti omaksumaan C#:n ja kyseisen ohjelmointikielen opiskelu painottui vahvasti syntaksin opettelemiseen, joka muistattaa hyvin pitkälti C++:n syntaksia. Suurimman ajan käytin .NET Framework kirjaston opiskeluun, joka tarjosi todella nopean ohjelmiston kehityksen, koska kirjaston luokat ovat jo valmiiksi hyvin suunniteltuja ja testattuja. Aikaa C#:n ja .NET Framework kirjaston eri osien opiskeluun käytin ennen toteutuksen aloittamista noin 60 tuntia.

Windows Presentation Foundationin opiskeluun varasin alun perin aikaa noin 40 tuntia, jotka käytin opiskeluun ennen opinnäytetyön toteuttamisen aloittamista. Nopeasti opinnäytetyön käyttöliittymän suunnittelun ja toteuttamisen jälkeen huomasin, että opiskelua on pakko jatkaa vielä. WPF:n opiskelu jatkuikin hyvin

pitkälti koko opinnäytetyön tekemisen ajan, koska WPF:n tarjoamat mahdollisuudet ovat valtavat. Arkkitehtuurin suunnittelun aloittamisen aikoihin tutustuin vielä Model-View-ViewModel-arkkitehtuurimalliin, joka on erityisesti Windows Presentation Foundationin kehittäjien suosiossa. Arkkitehtuurimallin opiskelu lisäsi entisestään opiskelumäärää, mutta tarjosi huomattavat edut ohjelman ylläpidettävyyden ja lisäkehityksen kannalta.

Toisessa palaverissa keskityttiin enemmän ohjelman toimintaan ja sen tarvitsemiin tietokantakyselyihin. Palaverissa huomattiin, että tietokantakyselyt ovat hyvin yksinkertaisia, jonka vuoksi Entity Frameworkin opiskeluun varasin vain noin 20 tuntia. Tämä aikataulu oli lähimpänä toteutunutta opiskelumäärää ja jälkikäteen arvioituna vähemmällä Entity Frameworkin opiskelulla olisi saatu tietokantakyselyt hoidettua mallikkaasti.

8.3 Toteutus

Toteutus sisältää ohjelman suunnittelun, ohjelmoinnin sekä testaamisen. Ohjelman toteutustavaksi oli valittu ketterä kehitysmenetelmä, jonka vuoksi ohjelmaa tehtiin aina lyhyissä iteraatioissa, jotka saattoivat olla lyhyimmillään noin päivän mittaisia.

Tämän kehitysmenetelmän takia monet eri työtehtävät menivät usein päällekkäin, jotta pystyttiin nopeasti vastaamaan asiakkaan haluamiin muutoksiin ja lisäominaisuuksiin. Yleensä asiakkaalle annettiin aina uusi versio ohjelmasta käyttöön, kun uudet ominaisuudet oli suunniteltu, ohjelmoitu sekä testattu mahdollisimman kattavasti. Uusien versioiden aikaväli oli lyhyimmillään muutaman tunnin mittaisia, koska uudet päivitykset olivat asiakkaan kannalta erittäin tärkeitä.

Ohjelman toteutus alkoi käyttöliittymän suunnittelusta, jossa ensimmäiseksi hahmoteltiin ohjelman pääikkuna. Pääikkunan käyttöliittymän saavutettua asiakkaan hyväksynnän ja lähes lopullisen ulkoasun, alkoi ohjelman arkkitehtuurin suunnittelu. Koska Windows Presentation Foundationin tarkoitus on erottaa käyttöliittymä ja ohjelman toiminta toisistaan, niin arkkitehtuurissa tuli ottaa huomioon ohjelman toimintojen toimivuus, vaikka käyttöliittymää muokattaisiin.

Arkkitehtuuriksi valittiin Model-View-ViewModel, jonka avulla käyttöliittymä ja ohjelman toiminta pystyttiin erottamaan toisistaan helposti. Arkkitehtuuri tarjoaa myös erittäin kätevän tavan laajentaa ohjelman toiminnallisuutta, koska arkkitehtuurin perustana on luokka, johon Windows-luokan DataContext-ominaisuus viittaa. DataContext on System.Object-luokan olio, jolloin ohjelman logiikkakoodin, C#:n, puolella käytettäessä DataContextille täytyy aina tehdä tyyppimuunnos ennen sen ominaisuuksien käyttämistä.

Käyttöliittymän puolella DataContextin käyttö on huomattavasti yksinkertaisempaa, eikä tyyppimuunnoksia tarvitse tehdä. Tämä mahdollistaa myös samojen toimintojen käyttämisen useissa eri käyttöliittymän elementeissä ilman, että ohjelman koodiin tarvittaisi tehdä muutoksia.

Ohjelmointi aloitettiin jo ennen kuin arkkitehtuuria oli täysin valittu, että asiakkaalle pystyttäisiin mahdollisimman nopeasti näyttämään luonnosmuotoisesti ohjelman toimintaa. Arkkitehtuurin valinnan jälkeen ohjelmaan siihen mennessä kirjoitettu koodi muutettiin toteuttamaan Model-View-ViewModel-arkkitehtuuria.

Ohjelman toteutusvaiheeseen oli varattu noin 300 tuntia, joka piti paikkansa erittäin hyvin ottaen huomioon, että toteutusvaiheessa opiskeluun tuli vielä käytettyä monta kymmentä tuntia. Toteutusvaiheen loppupuolella tuli opiskeltua muun muassa Microsoftin Windows Communication Foundation -teknologiaa, joka kuuluu ohjelman jatkokehitysvaiheeseen, eikä määriteltyyn opinnäytetyön laajuuteen. Kaikki toteutuksen asiat huomioon ottaen, aikaa toteutusvaiheeseen kului 306 tuntia.

9 Ohjelman esittely

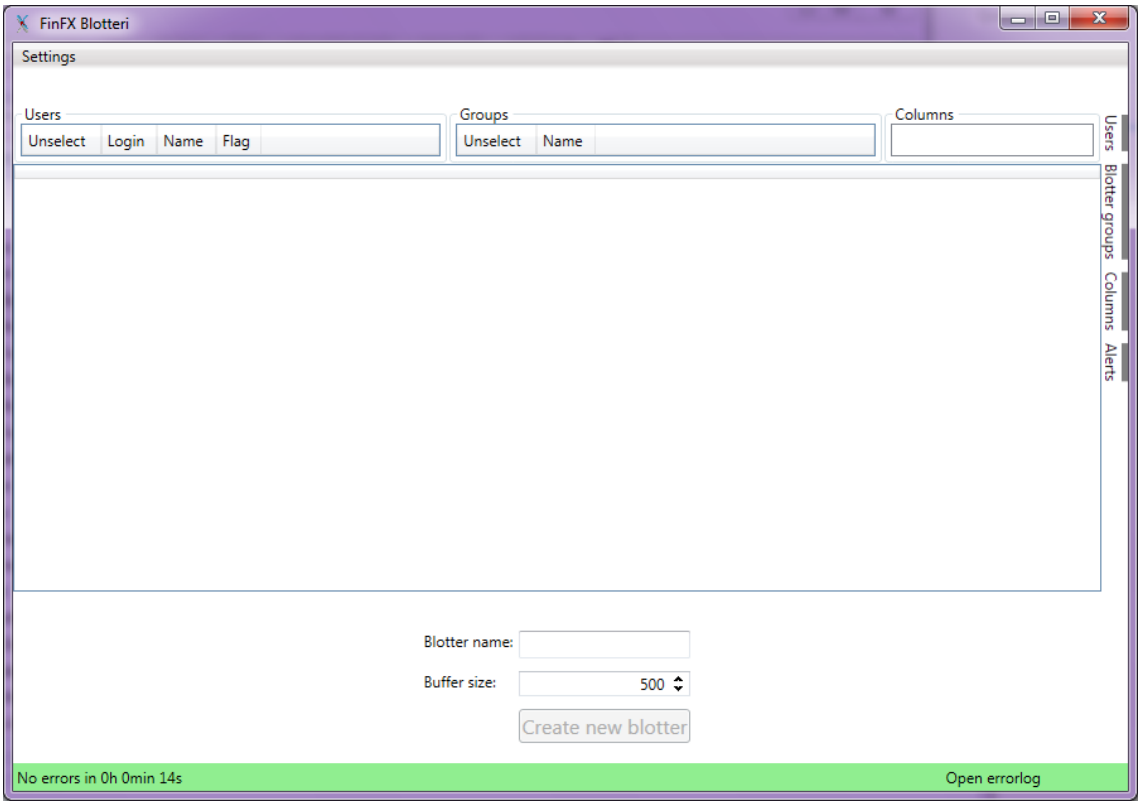
Tässä luvussa käydään läpi ohjelman käyttöliittymä sekä toiminnallisuus. Ohjelman toimintoja ovat muun muassa kauppojen suodatus valittujen kriteerien perusteella, ikkunoiden tallennus ja lataus.

9.1 Käyttöliittymä

Käyttöliittymän suunnittelun onnistuminen on ohjelman käytettävyyden kannalta erittäin tärkeää. Tulevaisuudessa mahdolliset laajennukset ohjelman toimintaan tuli myös ottaa huomioon jo suunnitteluvaiheessa, että toimintojen lisääminen ei vaatisi nykyisen käyttöliittymän merkittävää muokkaamista. Ohjelman käyttöliittymä on jaettu kolmeen erilliseen käyttöliittymään, jotka käydään tarkemmin seuraavissa aliluvuissa.

9.1.1 Pääikkuna

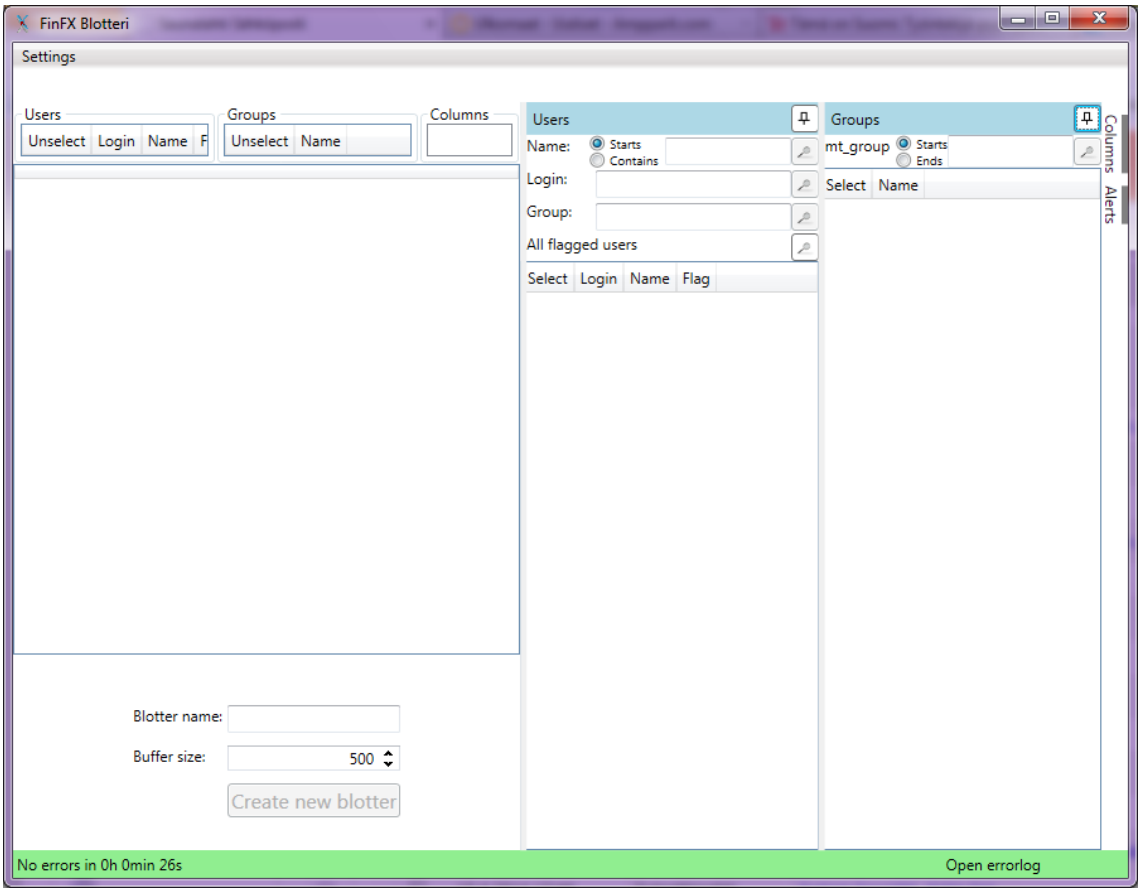
Ohjelma aukeaa kuvan 15 mukaiseen näkymään. Pääikkunan tehtävä on luoda uusia ikkunoita, joissa seurataan kauppoja valittujen kriteerien perusteella. Näitä kriteerejä voivat olla muun muassa käyttäjät tai ryhmät, johon käyttäjiä kuuluu.



Kuva 15. Pääikkunan näkymä.

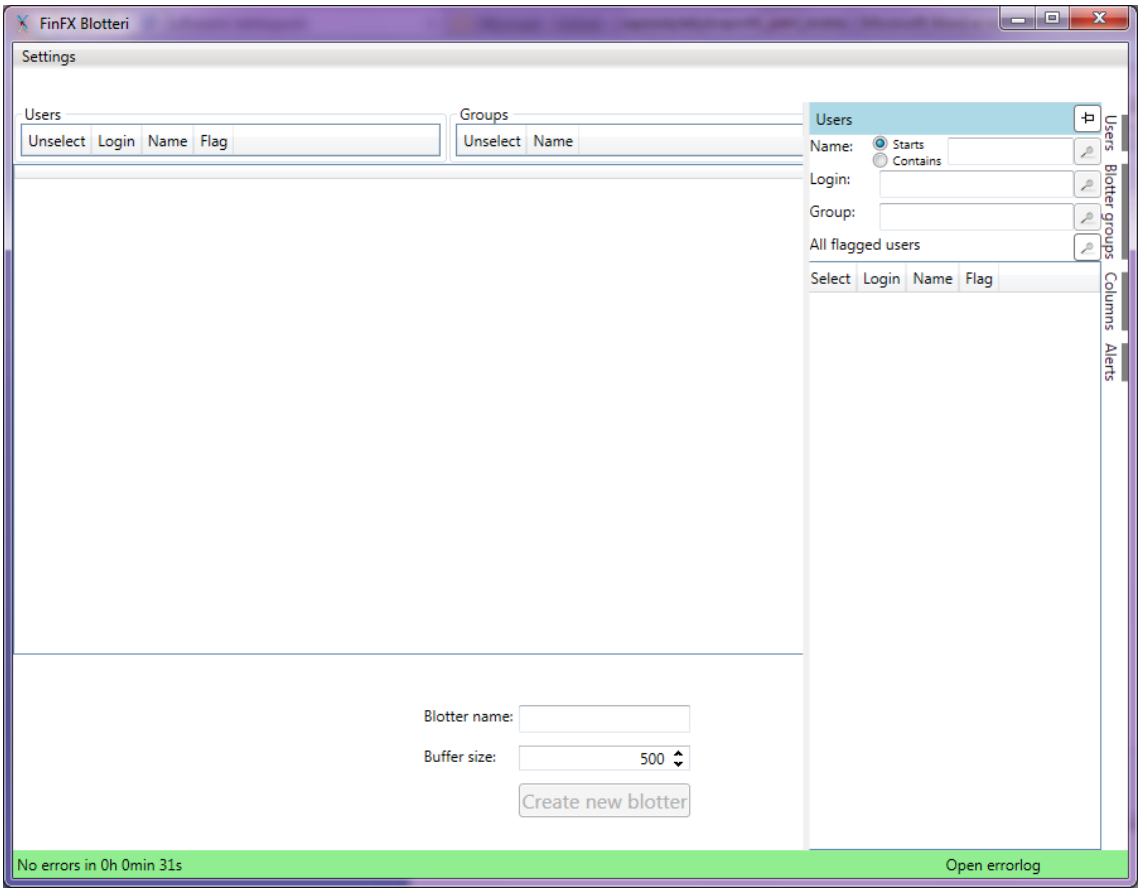
Oikeassa sivupalkissa on määriteltynä eri kriteerien valintaikkunat, jotka aukeavat dynaamisesti pääikkunan päälle niitä painaessa (kuva 15). Useita valintaik-

kunoita voi avata toistensa viereen, jos ne ensin kiinnitetään pääikkunaan. Tämä käyttöliittymän rakenne mahdollistaa tulevaisuudessa uusien valintaikkunoiden yksinkertaisen lisäämisen ohjelmaan ilman, että käyttöliittymää tarvitsee muokata. Kuva 16 näyttää valintaikkunoiden toimintaperiaatetta.



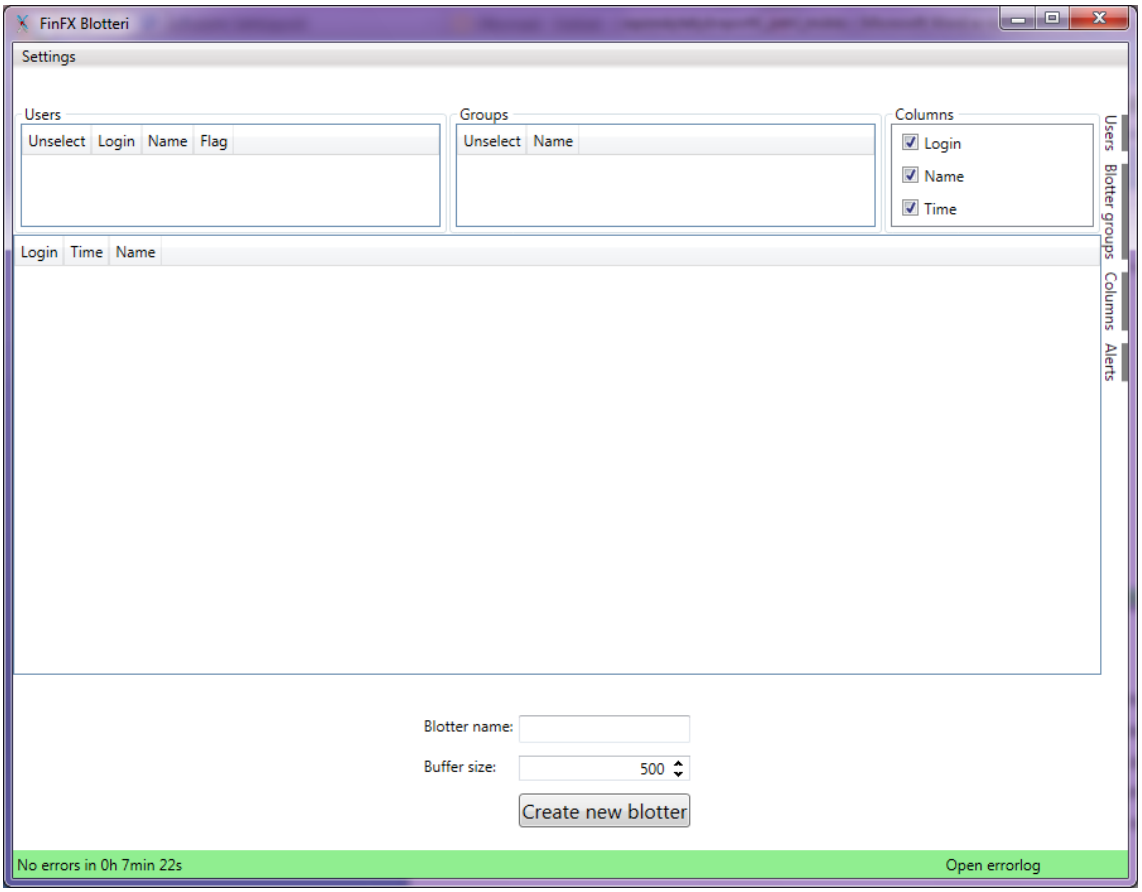
Kuva 16. Valintaikkunoiden toimintaperiaate.

Valintaikkunoiden leveyttä pystyy myös säätämään vetämällä niitä niiden oikeasta reunasta. Ilman manuaalista leveyden säätöä leveys mukautuu aina automaattisesti tarvittavaan leveyteen. Valintaikkunoita pystyy myös avaamaan ilman niiden kiinnittämistä pääikkunaan, jolloin ne avautuvat pääikkunanäkymän päälle. Kuva 17 esittää ei kiinnitetyn valintaikkunan toimintaa.



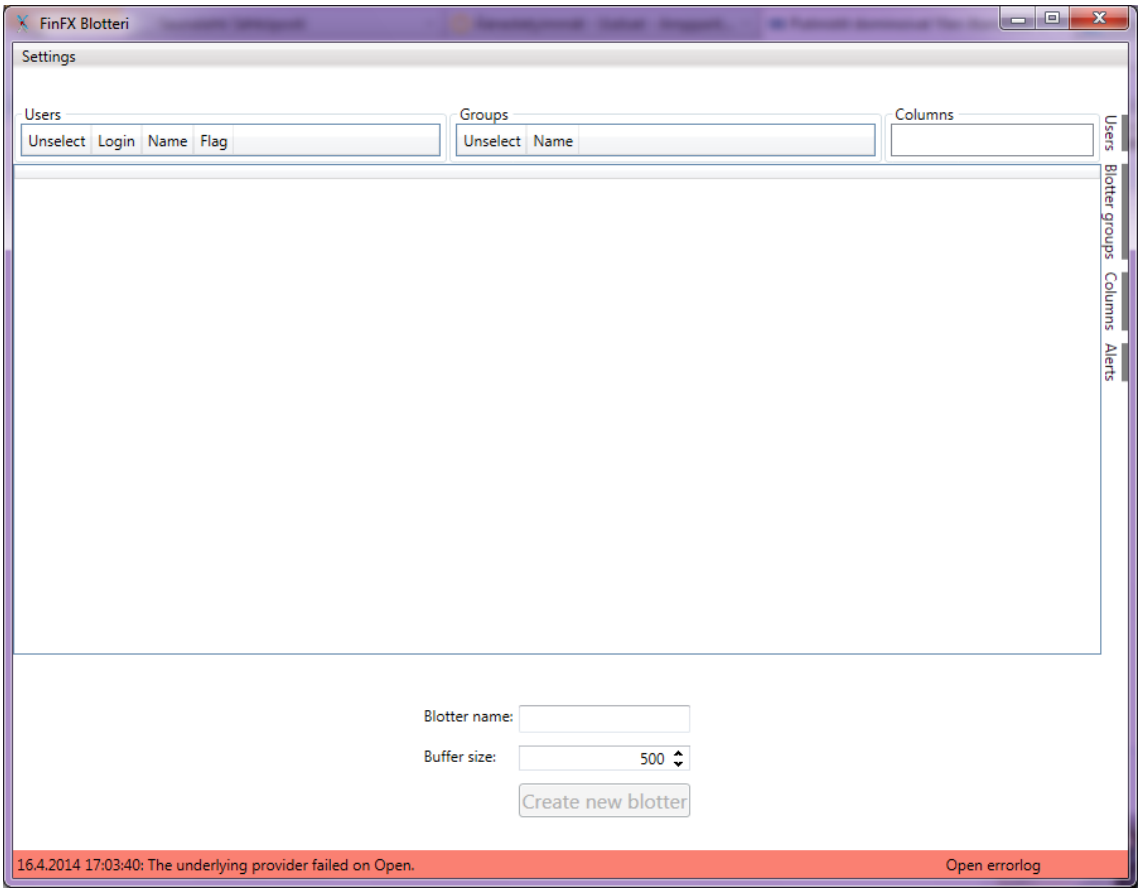
Kuva 17. Valintaikkunan toiminta ilman kiinnitystä pääikkunaan.

Pääikkunan keskellä on esinäkymä uudesta luotavasta kauppojen seurantaikkunasta, joka näyttää kauppoja valittujen kriteerien perusteella. Oletuksena esinäkymässä listataan kaikki asiakkaiden tekemät kaupat, ellei käyttäjä ole valinnut tiettyjä kriteerejä, joita kauppojen on täytettävä tai seurattavia henkilöitä tai ryhmiä. Kuva 18 näyttää esinäkymän, johon on valittuna näytettäviksi tiedoiksi henkilön käyttäjätunnus, aika sekä nimi kaikille uusille kaupoille.



Kuva 18. Esinäkymä, johon on valittuna näytettäviä tietoja.

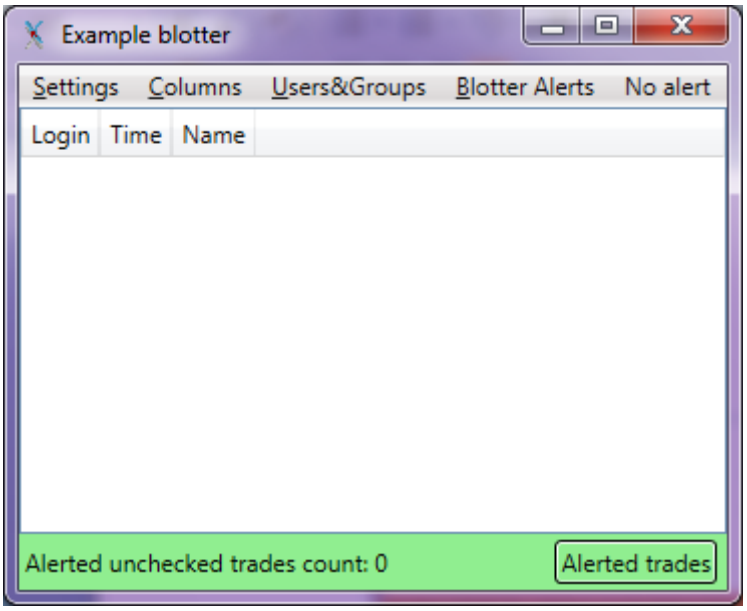
Ohjelman pääikkuna kertoo alapalkissa käyttäjälle ohjelman sen hetkisen tilanteen, eli onko virheitä tapahtunut ohjelman suorituksen aikana. Virheet, jotka yleensä ilmaantuvat ohjelman suorituksen aikana ovat tietokantayhteyksiin liittyviä, koska tietokannalla voi olla samalla hetkellä useita eri käyttäjiä. Ohjelman toiminta on täysin riippuvainen toimivista tietokantayhteyksistä, niin tämän tiedon välittäminen ohjelman käyttäjälle on erittäin tärkeää. Esimerkiksi, jos ohjelma ei ole välittänyt uusia kauppatietoja viiteen minuuttiin, on käyttäjän kannalta erittäin tärkeää tietää mahdollinen tietokantayhteyksiin liittyvä syy. Kuva 19 ilmoittaa käyttäjälle tietokantayhteyteen liittymästä virheestä.



Kuva 19. Virheilmoitus tietokantayhteydestä.

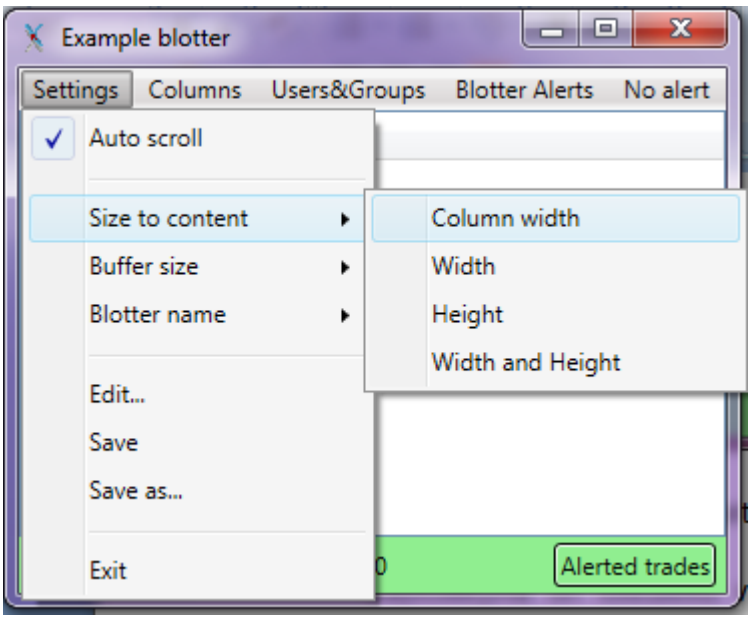
9.1.2 Kauppojen seurantaikkuna

Kauppojen seurantaikkunan tarkoitus on antaa ohjelman käyttäjälle mahdollisuus seurata useita eri kriteereillä olevia kauppoja yksinkertaisesti ja keskitetysti. Tämän ansiosta on mahdollista pitää auki samanaikaisesti useita eri kauppojen seurantaikkunoita. Kuva 20 näyttää ikkunan näkymän valittujen tietojen perusteella.



Kuva 20. Kauppojen seurantaikkuna.

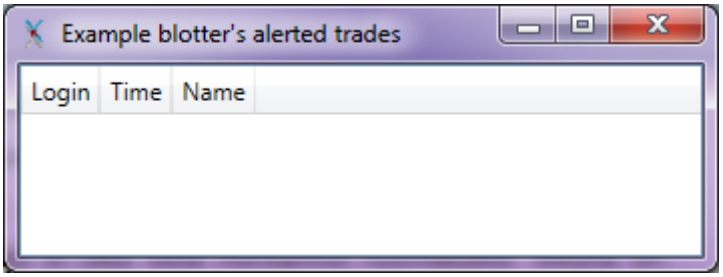
Käyttöliittymä on toteutettu toimimaan valikkopalkin kautta, josta löytyvät kaikki pääikkunan ominaisuudet. Käyttöliittymän suunnittelu eroaa pääikkunan näkymästä, koska tavoitteena oli saada ikkunasta mahdollisimman tiivis kokonaisuus, että samalla ruudulla on mahdollisuus pitää useita kauppojen seurantaikkunoita auki. Kuva 21 näyttää tämän tiiviimmän käyttöliittymän kauppojen seurantaikkunalle.



Kuva 21. Kauppojen seurantaikkunan käyttöliittymän toiminta.

9.1.3 Hälytettyjen kauppojen seuranta

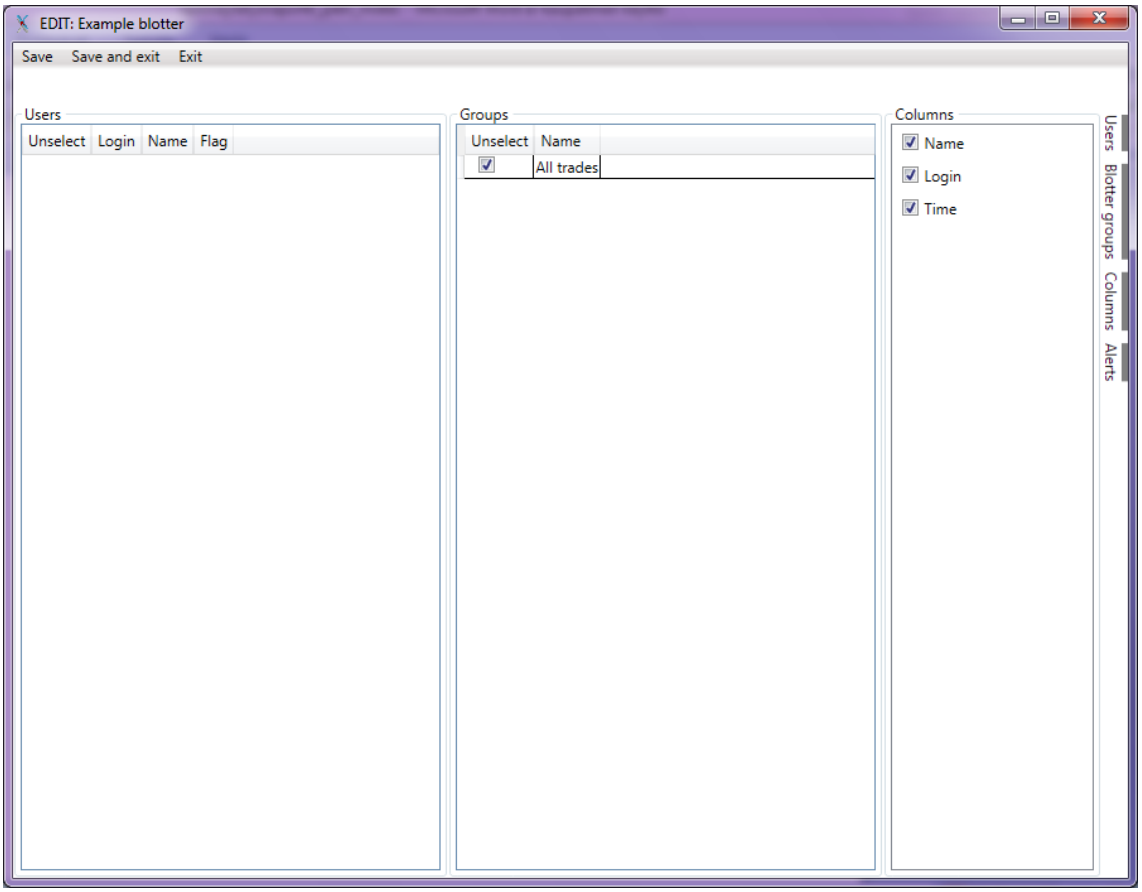
Ohjelma on suunniteltu toimimaan koko ajan taustalla, vaikka käyttäjä poistuisikin pitemmäksi aikaa tietokoneelta. Tämän takia jokaisella kauppojen seurantaikkunalla on mahdollisuus avata uusi ikkuna, joka esittää kaikki hälytyksen antaneet kaupat, joita käyttäjä ei ollut vielä merkannut huomatuiksi. Ikkuna on hyvin yksinkertainen ja mahdollistaa pelkästään kauppojen listaamisen ilman mitään omaa toiminnallisuutta johtuen siitä, että kyseessä on vain aputyökalu kauppojen seurantaikkunalle. Vaikka toiminta on hyvin yksinkertainen, niin on erittäin tärkeää pystyä listaamaan hälytyksen antaneet kaupat yhteen ikkunaan, koska yhdessä sekunnissa voi tulla useita kymmeniä kauppoja, joista yksi tai kaksi pelkästään antaa hälytyksen. Kuva 22 näyttää hälytettyjen kauppojen yksinkertaisen käyttöliittymän.



Kuva 22. Hälytyksen antaneiden kauppojen listaus.

9.1.4 Muokkausikkuna

Ohjelman käyttöliittymän suunnittelussa pyrittiin mahdollisimman johdonmukaiseen ja helposti ymmärrettävään käyttöliittymään. Muokkausikkunan toiminnallisuus vastaa hyvin pitkälti ohjelman pääikkunan toiminnallisuutta, jonka takia käyttöliittymä on lainattu suoraan pääikkunasta. Kuva 23 näyttää muokkausikkunan käyttöliittymän.



Kuva 23. Muokkausikkunan käyttöliittymä.

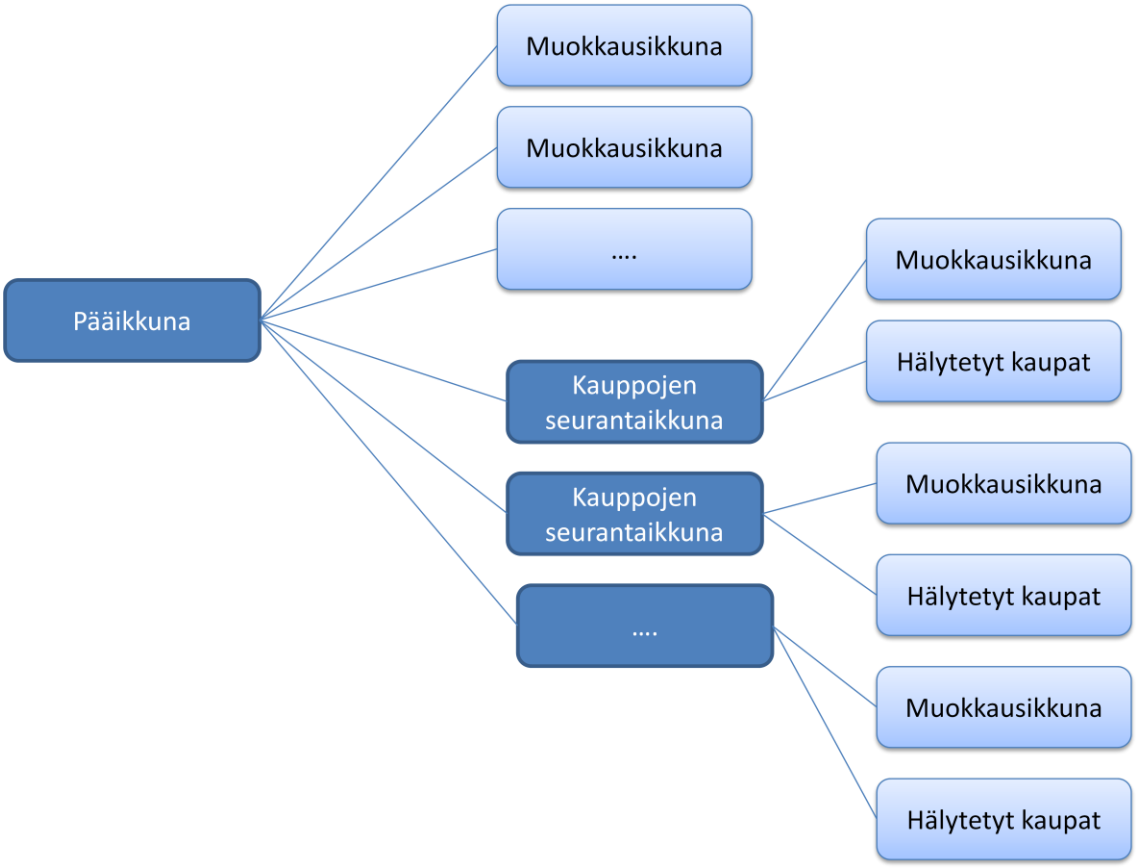
Käyttöliittymän toiminnallisuus sivupalkin osalta vastaa täysin pääikkunan sivupalkin toiminnallisuutta. Käyttäjällä on mahdollisuus kiinnittää valintaikkunoita käyttöliittymään tai avata niitä yksikerrallaan nopeasti. Käyttöliittymän samankaltaisuus pääikkunan kanssa mahdollistaa ohjelman käyttäjille nopean ja vaivattoman muokkauskokemuksen, kun on vain oppinut käyttämään ohjelman pääikkunaa.

9.2 Ohjelman toiminta

Tässä luvussa käydään läpi ohjelman toimintaa sekä toimintoja. Ohjelman tärkein toiminto on hakea uudet kauppatiedot tietokannasta ja jakaa ne eteenpäin avatuille kauppojen seurantaikkunoille, jotka osaavat valita kyseiseen ikkunaan kuuluvat kaupat.

9.2.1 Ikkunoiden hallinta

Ohjelman ajoaikana voi samanaikaisesti olla auki useita ikkunoita, joita pitää pystyä hallitsemaan, jos esimerkiksi ohjelman pääikkuna suljetaan. Jokainen ikkuna aina hallitsee sen omia lapsi-ikkunoita. Esimerkiksi ohjelman pääikkuna hallitsee kaikkia avattuja kauppojen seurantaikkunoita sekä pääikkunasta avattuja muokkausikkunoita. Kauppojen seurantaikkuna puolestaan hallitsee siitä avattuja ikkunoita, muokkausikkunaa ja hälytettyjen kauppojen seurantaikkunaa. Kuva 24 esittää ohjelman ikkunoiden hallintaa



Kuva 24. Ohjelman ikkunoiden hallintaperiaate.

Ikkunoiden hallintaperiaate on, että ikkunan sulkeutuessa se sulkee kaikki omat lapsi-ikkunansa. Tällä tavalla ohjelma aina varmistaa, että sulkeutuessa kaikki ohjelmasta käynnistetyt ikkunat tulevat sulkeutumaan.

Esimerkiksi, jos on pääikkuna, josta on avattu yksi muokkausikkuna sekä yksi kauppojen seurantaikkuna. Kauppojen seurantaikkunasta on lisäksi avattuna vielä yksi hälytettyjen kauppojen seurantaikkuna. Kun käyttäjä sulkee ohjelman

pääikkunan, niin se lähettää käskyn muokkausikkunalle ja kauppojen seurantaikkunalle, että niiden tulee myös sulkeutua. Kauppojen seurantaikkuna lähettää saman käskyn eteenpäin hälytettyjen kauppojen seurantaikkunalle, joka myös sulkeutuu.

9.2.2 Kauppojen hakeminen

Ohjelman tärkein ominaisuus on pystyä hakemaan uudet kaupat tietokannasta luotettavasti tietyin määräajoin. Tämä on määritelty ohjelmassa. Oletuksena on sekunnin välein tapahtuva kauppojen hakeminen tietokannasta, jolloin käytetään viimeksi haettujen kauppojen viimeisiä alkamis- ja sulkemisaikoja tarkastamaan onko tietokantaan lisätty uusia kauppoja.

Tietokantaan tulevat päivitykset saapuvat useammasta eri tietokannasta ja tietokanta mistä ohjelma hakee tietonsa sisältää kaikki kauppojen tiedot. Toiminnan suunnittelussa oli erittäin tärkeää huomioida, että tietokantaan saatetaan lisätä uutta tietoa esimerkiksi ajanhetkellä 15:50:42, jonka jälkeen ohjelma hakee tietoa sekunnilleen samalla hetkellä, jonka jälkeen tietoa lisätään uudelleen tietokantaan ajanhetkellä 15:50:42 ajassa. Koska ohjelman on aina toimittava luotettavasti, niin ohjelman on aina pidettävä muistissa viimeksi haettujen kauppojen tietoja, joita jokaisen tietokantakyselyn jälkeen verrataan uusiin palautettuihin kauppoihin.

Edellä mainitun syyn vuoksi tietokantakyselyssä joudutaan aina hakemaan samat kauppatiedot, joiden alkamis- tai päättymisaika vastaa viimeisimpien avattujen tai suljettujen kauppojen aikoja. Tämän vuoksi kauppatietoja haetaan mahdollisesti useampaan kertaan ohjelman suorituksen aikana, jolloin ohjelman on osattava aina suodattaa nämä kaupat pois, kun siirrytään tiedon jakamiseen auki oleville ikkunoille.

Ohjelman määrittelyvaiheessa pidettiin tärkeänä, että ohjelmaan pystyy myöhemmässä vaiheessa lisäämään uusia tiedonlähteitä, jotka hakevat tietoa tietokannasta tietyin väliajoin. Vastaamaan tätä tarvetta luotiin abstrakti luokka, jonka jäsenmuuttujana on .NET Frameworkin System.Timers.Timer-luokka. Kyseisellä Timer- luokalla on tapahtuma, joka tapahtuu aina sille määritellyn ajan välein. Kuva 25 näyttää UML-kaavion kyseisestä luokasta.

Kuva 25. Abstrakti luokka, josta kauppojen hakemista varten luotu luokka on periytetty.

OnTimedSearch on abstrakti metodi, jonka jokaisen periytyvän luokan on määriteltävä. Tämä metodi tulee vastaamaan kutsuttavaa metodia, joka tapahtuu aina määritellyin aikavälein. Kauppojen hakemistoiminnossa kyseiseen OnTimedSearch-metodiin on siis määritelty hakemaan kaikki uudet kaupat tietokannasta sekä jakamaan ne eteenpäin aukioleville kauppatietoja tarvitseville ikkunoille.

9.2.3 Kauppojen vastaanottaminen

Kun ohjelma on jakanut kaupat eteenpäin niitä hyödyntäville ikkunoille, on niiden vuoro suodattaa pois niille kuulumattomat kaupat. Ohjelmassa on mahdollista suodattaa kauppoja pois listauksesta useilla eri ehdoilla. Jokaisesta suodattimesta on luotu oma luokka, joka on periytetty yhteisestä kantaluokasta.

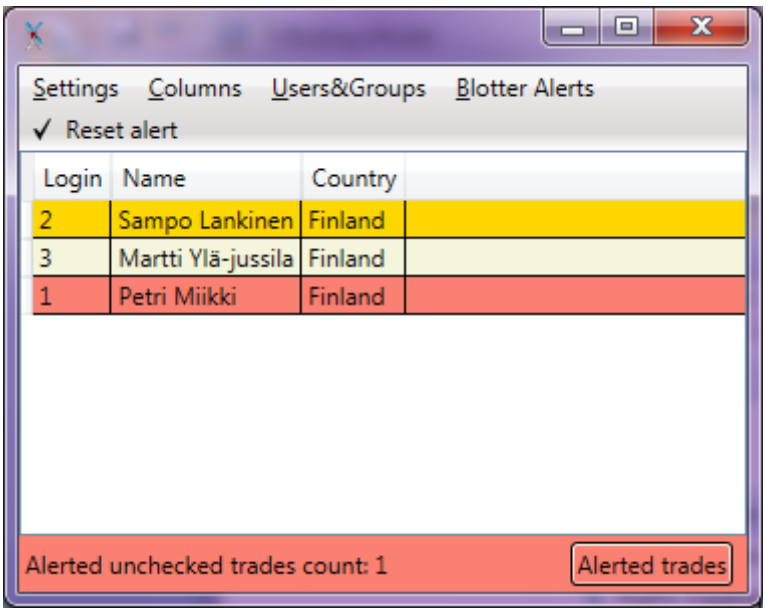
Käyttöliittymän puolella ohjelman käyttäjä pystyy valitsemaan ja määrittelemään arvon tai arvoja valitulle suodattimelle. Kun suodatin on valittu merkitseväksi tekijäksi, niin ohjelma osaa automaattisesti aina tarkistaa, että uudet kauppatiedot täyttävät valittujen suodattimien kriteerit. Jos uusi kauppatieto täyttää kaikki valitut kriteerit, niin kauppa tullaan näyttämään sitä hyödyntävässä ikkunassa. Jos yksikin suodatinkriteeri jää täyttymättä uuden kaupan kohdalla, niin kauppa suodatetaan pois. Myöhemmässä vaiheessa lisätyt suodattimet poistattavat jo lisätyt kaupat, jos ne eivät täytä uutta suodatinkriteeriä. Ohjelma näyttää aina uuden kaupan, jos ohjelman käyttäjä ei ole valinnut yhtään suodatin kriteeriä.

9.2.4 Hälytyksen antaminen

Ohjelman toiminnassa hälytettyjen kauppojen näyttäminen on yksi tärkeimmistä ominaisuuksista sekä syistä miksi FinFX Trading Oy:lle syntyi tarve ohjelmalle. Ohjelman käyttäjät pystyvät keskitetysti liputtamaan asiakkaitaan, jolloin liputettujen asiakkaiden kaupat aiheuttavat aina hälytyksen.

TradeInformation on luokka, joka vastaa kauppatietojen hallinnasta ja muunnoksista, joita tarvitaan tietokannasta haettavien tietojen muuttamiseen luettavampaan muotoon ohjelman käyttäjän kannalta. Luokassa on luettelointi, jota käytetään määrittelemään, onko kyseinen kauppa aiheuttanut hälytyksen, onko hälytetty kauppa merkitty huomatuksi tai kauppa on normaali, eli ei ole aiheuttanut hälytystä.

Kaupan esitystapa ilmoittaa ohjelman käyttäjälle sen tilan. Kuva 26 esittää kauppojen esitystapaa. Ylin kauppa on hälytyksen antanut kauppa, joka on merkitty huomatuksi. Keskimmäinen on normaalitilassa oleva kauppa, eli kauppa ei ole aiheuttanut hälytystä missään vaiheessa. Alin kauppa on hälytyksen antanut kauppa, jota ohjelman käyttäjä ei ole vielä merkinnyt huomatuksi.



Kuva 26. Esimerkki kauppojen hälytystiloista.

Kuvassa 26 näkyy myös, kuinka käyttöliittymä muuten ilmoittaa, että kauppojen seurantaikkunassa on hälytyksen antaneita, ei huomattuja kauppoja. Valikko

palkissa ”No Alerts”-otsikko muuttuu ”Reset alert”-muotoon, joka lisäksi myös merkitään valituksi. Painamalla ”Reset alert”-valikon nappia, ohjelma muuttaa hälytyksen antaneet kaupat huomatuiksi, jolloin niiden taustaväri muuttuu. Ikkunan alapalkki kertoo käyttäjälle, kuinka monta huomaamatonta hälytyksen antanutta kauppaa ikkunassa on ja yksinkertaisen tavan katsoa nopeasti kaikki hälytetyt kaupat ”Alerted trades”-nappia painamalla.

Hälytyksien antaminen on kaksivaiheinen ohjelman toiminnassa. Kauppojen hakeminen-toiminto tarkastaa jokaisen uuden kaupan kohdalla onko kaupan asiakas liputettu. Mikäli asiakas on liputettu, niin kaupan hälytyksen tilaksi määritellään ”Raised”. Kauppojen vastaanottaminen on toinen vaihe, jossa tarkistetaan aiheuttaako kauppa hälytyksen kyseiselle ikkunalle valittujen kriteerien perusteella.

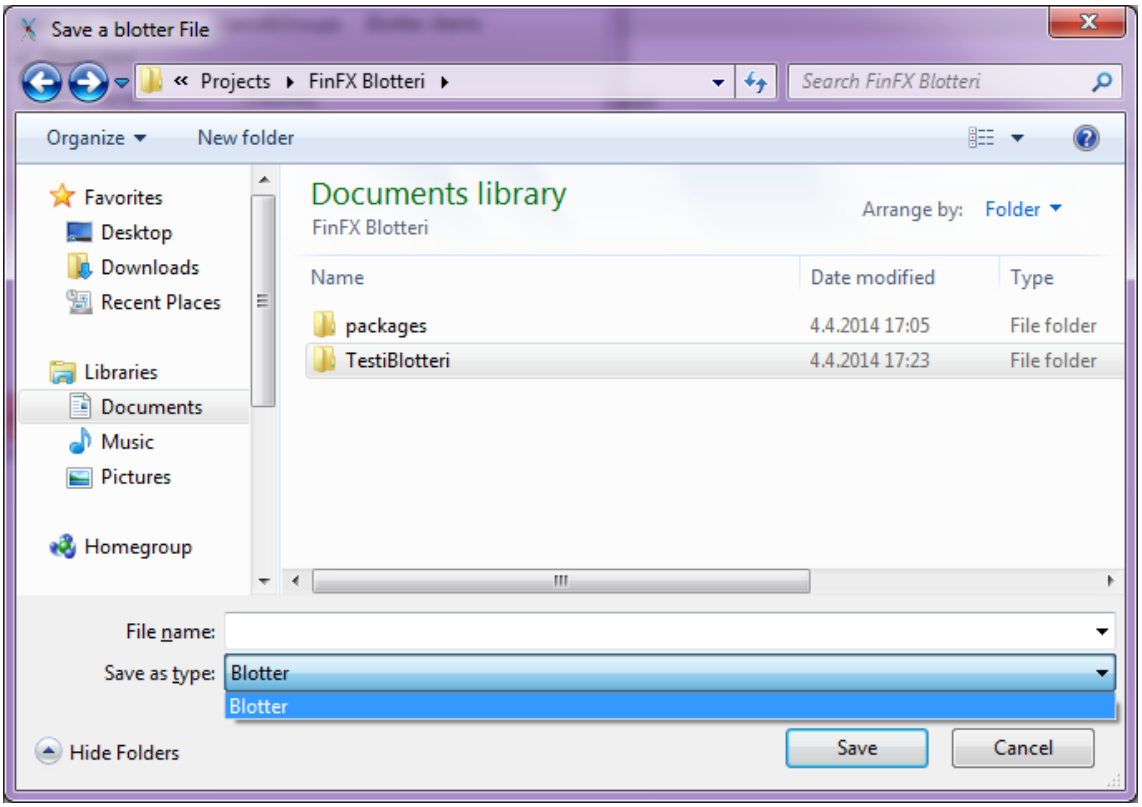
9.2.5 Tallentaminen ja lataaminen

Ohjelman tarkoituksena on seurata mahdollisimman tehokkaasti FinFX Trading Oy:n asiakkaiden tekemiä kauppoja. Helppokäyttöisyys oli toinen vaatimus ohjelman toiminnalle, ettei ohjelman käyttäjä joutuisi jokaisen käynnistyksen yhteydessä luomaan uudestaan ja uudestaan samoja kauppojen seurantaikkunoita samoilla kriteereillä. Vastaamaan tätä kriteeriä seurantaikkunoiden tallentaminen ja lataaminen on automatisoitu sarjallistamalla oliokaaviot.

Tallennusominaisuus osaa tallentaa kauppojen seurantaikkunan juuri siinä tilassa kuin se tallennushetkellä on. Jos käyttäjä on esimerkiksi leventänyt näytettävien kauppatietojen otsikoita, niin otsikoiden leveys tallennetaan siinä muodossa, tai jos käyttäjä on siirtänyt kauppojen seurantaikkunaa ruudulla, niin lataamisvaiheessa se avataan tallennushetkellä sijainneeseen paikkaan. Sijainnin lataamisessa ohjelma osaa laskea, että ikkunan sijainti tulee olemaan ruudulla, koska on mahdollista, että tallennushetkellä on ollut käytössä useampi ruutu samanaikaisesti ja lataamishetkellä ruutujen määrä on vähentynyt. Ainoastaan jo näytettyjä kauppoja ei tallenneta, koska ohjelman toimintaperiaate on näyttää reaaliaikaisesti uusia kauppoja.

Käyttöliittymän suunnittelussa on hyvä periaate tehdä asiat aina käyttäjille jo tutuilla tavoilla, jonka takia kauppojen seurantaikkunoiden tallentaminen ja la-

taaminen tapahtuu Microsoft Windowsin API:n kautta. Tämä mahdollistaa myös ohjelman uusille käyttäjille helpon ja tutun käyttöliittymän. Kuva 27 näyttää ohjelman tallentamisikkunan. Tiedostomuoto tallennuksissa ja latauksissa on aina .blotter ja ohjelma sallii pelkästään tämän tiedostotyypin.



Kuva 27. Tallentamisen käyttöliittymä.

Ohjelmassa on automaattinen kauppojen seurantaikkunoiden tallennus ohjelman sulkeutuessa. Automaattisesti tallennetut kauppojen seurantaikkunat aukeavat seuraavalla ohjelman käynnistyskerralla automaattisesti. Tavoitteena oli saattaa ohjelman päivittäinen käyttäminen mahdollisimman yksinkertaiseksi ja tehokkaaksi.

9.2.6 Fontin koon muuttaminen

Ohjelman toiminnan kannalta fonttikoon muuttaminen on erittäin tärkeää, koska ohjelmaa tullaan käyttämään erikokoisilla näyttöpäätteillä. Fontin muuttaminen mahdollistaa selkeän käyttöliittymän niin pienillä kuin isoilla näyttöpäätteillä.

Fontin muuttaminen tapahtuu ohjelman pääikkunan valikko-palkista. Fontin koon muuttaminen ohjelman koodissa tapahtuu samalla tavalla kuin ikkunoiden hallinta ohjelmassa. Ikkuna välittää aina tiedon sen lapsi-ikkunoille fonttikoon muutoksesta, jolloin jokainen ikkuna yksitellen reagoi tapahtumaan. Toisin sanoen, pääikkunan fonttikoon muuttaminen vaikuttaa myös ohjelman kaikkien auki olevien ikkunoiden fonttikokoa.

10 Yhteenveto

Opinnäytetyön tavoitteena oli saada suunniteltua, ohjelmoitua sekä testattua asiakkaan, FinFX Trading Oy:n, päivittäisiä työtehtäviä helpottava Trade Blotter-ohjelma. Opinnäytetyön puitteisiin rajattu ohjelma saatiin suunniteltua, ohjelmoitua ja testattua kattavasti. Asiakkaan palautteen perusteella ohjelmasta tuli juuri sellainen kuin asiakas oli toivonutkin ja se päätyi heidän päivittäiseen käyttöönsä.

Opinnäytetyön aikana opin todella paljon uusia asioita, joita pystyn tulevaisuudessa erittäin todennäköisesti vielä hyödyntämään. Ohjelmoinnin osalta ohjelma ei tuottanut sen suurempia haasteita, koska olin ennen opinnäytetyön aloittamista opiskellut ohjelmointia erityisesti C++:lla hyvin paljon. Ohjelmointiin liittyvät haasteet rajoittuivat pitkälti aina ohjelman logiikkaan, ei sinällään itse koodin tuottamiseen. Erityisesti ohjelmassa samanaikaisesti suoritettavat kaksi säiettä tuottivat paljon samanaikaisuusongelmia, jonka teoriaa olin aikaisemmin lukenut C++:n puolelta. Tämä tietotaito auttoi huomattavasti näiden ongelmien ratkomista sekä osasin ottaa huomioon monia samanaikaisuuteen liittyviä ongelmia, jotka olisin todennäköisesti muuten jättänyt huomioimatta.

Ohjelmaa tehdessäni opin käyttämään monia erilaisia valmiita ohjelmointikehyksiä. Esimerkiksi tietokantakyselyt tehtiin kokonaan Entity Frameworkilla ja ohjelman virhetilanteista kirjoitettavaan lokiin käytettiin Apachen log4net-ohjelmointikehystä. Näiden ohjelmointikehysten käyttäminen helpottaa myös ohjelman koodin ymmärtämistä, koska useat eri toiminnot on tehty abstraktilla tasolla.

Erityisesti ohjelmiston suunnittelutaitoni kehittyi, koska tutustuin useisiin erilaisiin ohjelmistoarkkitehtuureihin. Minulla ei juuri ollut aiempaa kokemusta ohjelmistoarkkitehtuureista, joten opinnäytetyö tarjosi erittäin hyvän tilaisuuden panostaa tähän osa-alueeseen.

Suurin osa-alue, jonka opinnäytetyössä pääsin opiskelemaan, oli käyttöliittymien tekemiseen tarkoitettu Windows Presentation Foundation. Aikaisempaa kokemusta minulla oli vain koulussa käytetystä ja myös Microsoftin kehittämästä Windows Forms -käyttöliittymäteknologiasta. Opinnäytetyötä tehdessä huomasin miten paljon kehittyneempi WPF on verrattuna vanhempaan Windows Formsiin ja tulevaisuudessa tuskin tulen enää käyttämään Windows Formsia.

Omasta mielestäni opinnäytetyö onnistui erittäin hyvin ja sitä tehdessä tuli esiin myös monia jatkokehitysajatuksia, jotka tulisivat parantamaan ohjelman toimintaa. Valitettavasti jatkokehitykseen ei ollut opinnäytetyön puitteissa mahdollisuutta, koska aikataulu ylittyi jo opiskelussa ja työosuudessa noin 20 tunnilla.

Jatkokehityksessä tulisi muuttaa hieman ohjelman luokkien rakennetta ja luoda niin sanottu palvelinohjelma. Palvelinohjelman olisi tarkoitus hakea tietokannasta kauppatiedot ja jakaa ne eteenpäin siihen yhteydessä oleviin ohjelmiin, jolloin tietokantaan suoritettaisiin aina vakiomäärä kyselyitä uusia kauppatietoja haettaessa, vaikka samanaikaisesti olisi käynnissä X määrä ohjelmia. Myös muutamia muita pienempiä jatkokehitysideoita syntyi, mutta ne ovat lähinnä kosmeettisia ja hyvin helppoja toteuttaa.

Kuvat

Lähteet

Agile Manifesto, Manifesto for Agile Software Development
http://agilemanifesto.org/. Luettu 27.4.2014

Agile Manifesto, Principles
http://agilemanifesto.org/principles.html. Luettu 27.4.2014

Albahari, J & Albahari, B. 2012. C# in a Nutshell. ISBN 978-1-449-32010-2

Birth of OO, The Simula Languages
http://www.olejohandahl.info/old/birth-of-oo.pdf. Luettu 25.4.2014

Champlain & Patrick. 2005. C# 2.0: Practical Guide for Programmers. ISBN: 0-12-167451-7

Clark, D. 2011. Beginning C# Object-Oriented Programming. ISBN-13 978-1-4302-3530-9

DACS, Agile Software Development
http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.201.2704&rep=rep1&type=pdf. Luettu 27.4.2014

FinFX Trading Oy.
https://www.finfx.fi/fi/tietoa-meista. Luettu 8.4.2014

Haikala, I & Mikkonen, T. 2011. Ohjelmistotuotannon käytännöt. Helsinki: Talentum Media Oy.

Jones, B. 2004. Sams Teach Yourself the C# Language in 21 Days. ISBN 0-672-32546-2

Lerman, J. 2010. Programming Entity Framework. ISBN 978-0-596-80726-9

Microsoft, .NET Framework
http://msdn.microsoft.com/en-us/library/zw4w595w%28v=vs.110%29.aspx. Luettu 23.4.2014

Microsoft, Attributes
http://msdn.microsoft.com/en-us/library/z0w1kczw.aspx. Luettu 24.4.2014

Microsoft, Code Snippets
http://msdn.microsoft.com/en-us/library/ms165392.aspx. Luettu 5.5.2014

Microsoft, Data Binding.
http://msdn.microsoft.com/en-us/library/ms752347%28v=vs.110%29.aspx. Luettu 10.4.2014

Microsoft, Entity Framework
http://msdn.microsoft.com/en-us/library/gg696172%28v=vs.103%29.aspx.

Luettu 23.4.2014

Microsoft, Implementing the MVVM Pattern
http://msdn.microsoft.com/en-us/library/gg405484%28v=pandp.40%29.aspx. Luettu 2.5.2014

Microsoft, IntelliSense
http://msdn.microsoft.com/en-us/library/hcw1s69b.aspx. Luettu 4.5.2014

Microsoft, LINQ
http://msdn.microsoft.com/en-us/library/bb397926.aspx. Luettu 22.4.2014

Microsoft, LINQ Syntax
http://msdn.microsoft.com/en-us/library/bb397947.aspx. Luettu 22.4.2014

Microsoft, Object Element Syntax
http://msdn.microsoft.com/en-us/library/ms788723%28v=vs.110%29.aspx. Luettu 29.4.2014

Microsoft, Serialization
http://msdn.microsoft.com/en-us/library/ms233843.aspx. Luettu 24.4.2014

Microsoft, Threading Model.
http://msdn.microsoft.com/en-us/library/ms741870%28v=vs.110%29.aspx. Lueattu 20.4.2014

Microsoft, Trees in WPF.
http://msdn.microsoft.com/en-us/library/ms753391%28v=vs.110%29.aspx. Luettu 10.4.2014

Microsoft, Types
http://msdn.microsoft.com/en-us/library/ms173104.aspx. Luettu 23.4.2014

Microsoft, Value Types
http://msdn.microsoft.com/en-us/library/s1ax56ch.aspx. Luettu 23.4.2014

Microsoft, Visual Studio
http://msdn.microsoft.com/en-us/library/ms165079.aspx. Luettu 4.5.2014

Microsoft, Windows Presentation Foundation.
http://msdn.microsoft.com/en-us/library/aa970268%28v=vs.110%29.aspx. Luettu 9.4.2014

Microsoft, XAML.
http://msdn.microsoft.com/en-us/library/ms752059%28v=vs.110%29.aspx. Luettu 9.4.2014

MSDN Magazine, WPF MVVM
http://msdn.microsoft.com/en-us/magazine/dd419663.aspx. Luettu 27.4.2014

Nathan, A. 2010. WPF 4 Unleashed. ISBN 978-0-672-33119-0

Prata, S. 2012. C++ Primer Plus 6th edition. ISBN-13: 978-0-321-77640-2

Sheldon, B & Hollis, B & Windsor, R & McCarter, D & Hillar, G & Herman, T. 2012. Profesional Visual Basic 2012 and NET 4.5 Programming. Indianapolis: John Wiley & Sons, Inc.

TIOBE, Index for April 2014
http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html. Luettu 25.4.2014